Conception d'une machine virtuelle pour les systèmes parallèles à diffusion

Robert Despons

THESE

présentée par

Robert DESPONS

pour obtenir le grade de **DOCTEUR**

de l'INSTITUT NATIONAL POLYTECHNIQUE de GRENOBLE

(Arrêté ministériel du 30 mars 1992)

Spécialité : **Informatique**

Conception d'une Machine Virtuelle pour les Systèmes Parallèles à Diffusion

Date de soutenance : 3 décembre 1996

Composition du jury :

Président :	**Paul JACQUET**
Rapporteurs :	**Ibrahima SAKHO**
	André SCHIPER
Examinateurs :	**Roland BALTER**
	Guy MAZARE
	Traian MUNTEAN

Thèse préparée au sein du Laboratoire Logiciels, Systèmes, Réseaux

«La machine arithmétique fait des effets qui approchent plus de la pensée que tout ce que font les animaux ; mais elle ne fait rien qui puisse faire dire qu'elle a de la volonté, comme les animaux.»
Blaise Pascal

Je dédie cette thèse à celles que j'aime et qui m'aiment

Remerciements

Je tiens ici à remercier tous ceux sans qui cette thèse n'aurait pu voir le jour.

Paul Jacquet, pour l'honneur qu'il m'a fait de présider mon jury de thèse.

Ibrahima Sakho et André Schiper, pour avoir accepté d'être les rapporteurs de mon travail; leurs remarques et suggestions ont largement contribué à l'amélioration de mon manuscrit.

Roland Balter et Guy Mazaré, pour l'intérêt qu'ils ont manifesté pour mon travail en acceptant, malgré leurs contraintes, de participer au jury.

Traian Muntean, pour avoir dirigé mes travaux de thèse, pour tous les efforts qu'il a fait afin de me procurer les moyens pour travailler, et surtout pour l'amitié qu'il a su insérer dans nos relations.

«Mon grand Léon», pour avoir été mon premier lecteur et correcteur, pour avoir partagé tant de choses, et surtout pour son amitié.

Tous les membres de notre équipe «SyMPA» pour toutes leurs contributions, leur aide, et leurs amitiés. Tout d'abord les anciens de Grenoble: Philippe, Leïla, François et Ahmed, Sylvie, Ghazali, Alba, Martine notre secrétaire, et plus particulièrement Harold pour tout ce que nous avons fait ensemble en marge de notre bureau commun; puis les nouveaux de Marseille: Tin, Stephane, Christophe, Pedro et Vincent.

Enfin, je tiens à remercier tous ceux qui m'ont soutenu, encouragé et supporté durant ces quatre ans.

Résumé

Dans les machines parallèles les performances des programmes posent de manière cruciale le problème de l'efficacité des communications dans les réseaux d'interconnexion des processeurs d'une machine sans mémoire commune. Les communications point-à-point ne sont qu'un cas très particulier des schémas de communications complexes utilisés par les applications. Les communications globales, basées sur la construction correcte de protocoles à diffusion, sont une classe de ces schémas de communication. Ce problème comprend deux aspects: l'acheminement des messages pour la diffusion et la construction de protocoles de communication/synchronisation inter-processus.

Nous considérons d'abord le problème de l'acheminement pour la diffusion, que nous construisons à partir d'une fonction de routage correcte pour des réseaux généraux de topologies quelconques. La famille d'algorithmes de diffusion obtenus s'adapte à la fois à la représentation de la fonction de routage, et à la topologie d'interconnexion entre processeurs. Un aspect de l'efficacité des algorithmes produits est l'espace mémoire nécessaire à une telle fonction de routage à diffusion. Nous développons des algorithmes qui requierent un espace mémoire constant et qui de plus, en utilisant une représention par intervalles de la fonction de routage, peuvent être intégrés dans un circuit routeur.

Nous nous intéressons ensuite à la construction de divers types de protocoles à diffusion (synchrone et asynchrone) et proposons une machine virtuelle parallèle à diffusion (PDVM). Cette machine virtuelle s'inscrit dans l'architecture du micro-noyau pour systèmes parallèles ParX, développé par notre équipe, qui offre un support d'exécution générique pour de multiples machines virtuelles. PDVM se présente sous la forme de deux de protocoles nécessaires à l'élaboration de la plupart des schémas de communication par diffusion. L'interface d'accès à ces protocoles permet de gérer des groupes de processus à diffusion toujours cohérents. Dans sa conception cette machine virtuelle est un support minimal pour implémenter efficacement et correctement les diverses interfaces et bibliothèques de communications globales pour les standards de programmation parallèle qui émergent (PVM, MPI, etc.).

L'ensemble des solutions proposées a été intégré dans le prototype de ParX; et leurs résultats d'évaluation de performances sont produits.

Mots clés: Systèmes parallèles, contrôle des communications, routage à diffusion, diffusion synchrone, diffusion asynchrone.

Abstract

Performances of programs on distributed memory parallel machines are highly dependent of the efficiency of interprocessor communications. Parallel programming environments often offer poor support for high level communication models. This thesis deals with high level group communications in such architectures. This problem has two main aspects: first, correct and efficient handling of message exchanges for diffusion between processors, second, construction of group communication protocols for processes.

We first address the problem of correct message passing within diffusion exchanges on top of efficient and correct routing functions. The proposed solution is general and is independent of the representation of the routing function, and it is independent from the topology of the interconnection network. Moreover constant memory space is required to prevent deadlocks in general networks. Futhermore, using interval labelling to represent the routing function, we show how our diffusion method can be integrated in a routing processor.

Secondly, we consider the construction of high level diffusion protocols (synchronous, asynchronous), and propose a diffusing virtual machine for the parallel generic kernel ParX developped by our group. Our virtual machine is based on two diffusion protocols which can be used to correctly and efficiently build most of existing global communication schemes. The protocol interface offers primitives for coherent management of process groups, message exchanges and control (broadcasting and scattering). Our virtual machine is designed to be a minimal support to efficiently and correctly implement different existing group communication interfaces and libraries (PVM, MPI, etc.).

The set of proposed solutions has been integrated in the prototype version of the ParX kernel, and some performances evaluated.

Keywords: Parallel systems, communication control, diffusion routing, synchronous diffusion, asynchronous diffusion.

Sommaire

Partie I : L'Emergence des Systèmes à Diffusion

Chapitre 1 : Introduction, des Architectures Parallèles aux Modèles à Diffusion

Alors que l'électronique des ordinateurs n'était constituée que de tubes à vide, les pionniers de l'informatique, dont entre autres David Slotnick [Slot82], pensaient déjà à des calculateurs parallèles. Malheureusement la technologie des tubes à vide ne permettait pas d'assembler de telles machines. Ce ne fut envisageable qu'au début des années 60 avec l'apparition de composants miniaturisés, moins coûteux, et dégageant beaucoup moins de chaleur, comme le transistor (inventé par Walter Brattain, John Bardeen et William Shockley en 1948, commercialisé en 1954 par Texas Instruments pour le transistor au silicium) puis le circuit intégré (inventé par Robert Noyce et Jack St. Clair Kilby en 1958, commercialisé en 1961 par Fairchild et Texas Instruments). Ainsi en 1964, moins de vingt ans après la réalisation de l'ENIAC (J. W. Mauchly et J. P. Eckert), le premier projet de conception d'un calculateur parallèle vit le jour à l'université de l'Illinois. Placé sous la direction de D. Slotnick, et avec le soutien de l'U.S. Air Force, de Burroughs et Texas Instruments, ce projet aboutira à la construction du premier cal-

culateur parallèle: l'ILLIAC IV. Orienté vers les applications scientifiques et conçu plus particulièrement pour le calcul matriciel et la résolution d'équations différentielles partielles, l'ILLIAC IV a trouvé en la météorologie un domaine d'utilisation privilégié.

Cette machine a ouvert la voie au développement des super-calculateurs et a inspiré les concepteurs d'architectures parallèles. Même si son coût a fortement limité son utilisation, le parallélisme s'est toujours avéré comme une alternative réaliste au modèle de von Neumann, et notamment lorsque de fortes puissances de calcul sont requises. Aujourd'hui, en dépit des fantastiques évolutions des composants qui font que les microprocesseurs actuels sont capables d'atteindre des vitesses de traitement très élevées (64.5 SPECint92 et 56.9 SPECfp92 pour le *Pentium*, environ 60 SPECint92 et 80 SPECfp92 pour le *PowerPC 601*, ou encore 84.4 SPECint92 et 127.7 SPECfp92 pour l'*Aplha 21064* [SezVau94]), les contraintes technologiques (capacité d'intégration, fréquence d'horloge, etc.) limitent grandement les perspectives d'augmentation des puissances de calcul de ces processeurs.

Or l'outil informatique s'est inséré, et continue à s'insérer, irrémédiablement dans de nombreux domaines, et d'une manière telle que toujours plus de capacités sont nécessaires: que ce soit pour des applications à caractère scientifique (météorologie, mécanique des fluides, etc.), pour les systèmes embarqués, pour les applications multimédia, le monde des images de synthèse et de la réalité virtuelle, ou même dans l'informatique de gestion. Pour toutes ces activités la puissance de calcul et la disponibilité des machines sont devenus des facteurs importants d'évolution. Bien sûr la technologie des semi-conducteurs peut encore avancer, d'autres peuvent émerger (la supra-conductivité ou l'optique par exemple), mais le parallélisme reste néanmoins une solution à cette quête, ne serait-ce que parce qu'il est capable de tirer profit de ces évolutions.

C'est d'ailleurs dans cette direction que se sont engagés de plus en plus de centres de recherche, de compagnies industrielles et de constructeurs de machines. Ainsi, depuis l'ILLIAC IV, les calculteurs parallèles se sont multipliés, selon différents types d'architecture, toujours plus puissants, et avec un nombre croissant de processeurs.

1.1 Les principales architectures parallèles

Nés de besoins propres au calcul scientifique où les mêmes opérations sont appliquées de multiples fois sur des données différentes, les premiers ordinateurs parallèles, dans le sillage de l'ILLIAC IV, ont souvent adopté une architecture orientée vers un parallélisme de données: l'architecture **SIMD**[1]. Conjointement à l'évolution des machines SIMD les ordinateurs à usage général ont introduit le parallélisme dans leur architecture. Cette intégration a pris diverses formes que l'on peut regrouper dans la famille **MIMD**[2]. Dans cette famille de machines, deux philosophies se distinguent: le **partage de mémoire** et l'**échange de messages**.

1.1.1 Architectures SIMD

Pour cette classe d'architectures le parallélisme qui est exploité est celui des données par rapport aux calculs. Dans ce mode de calcul une même instruction est appliquée simultanément sur plusieurs unités de données. Une caractéristique essentielle de ces architectures est de con-

1. Single Instruction stream - Multiple Data stream: flot d'instructions unique - multiples flots de données selon la taxonomie de Flynn [Flynn72].
2. Multiple Instruction stream - Multiple Data stream: multiple flots d'instructions - multiples flots de données [Flynn72].

server un **synchronisme** total au cours de l'exécution d'un programme. De ce principe plusieurs méthodes de réalisation ont été proposées: les *processeurs vectoriels*[1], les *processeurs pipe-line* et les architectures *SIMD «pures»*.

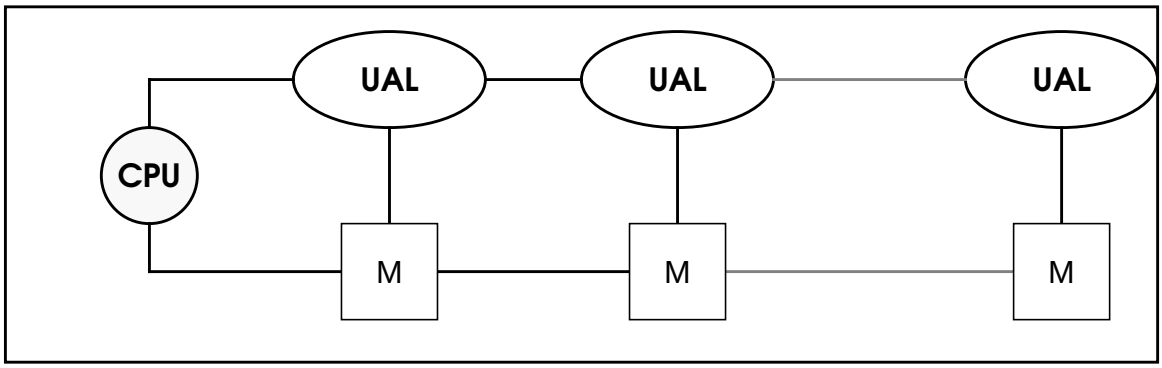


Figure 1.1 : architecture des processeurs vectoriels.

Dans un processeur vectoriel chaque instruction associe une opération à deux vecteurs d'opérandes. Pour exécuter de telles instructions le processeur possède plusieurs unités de calcul (UAL); et, une fois les vecteurs placés dans les registres des UAL et l'opération reconnue, chaque UAL exécute l'opération sur les opérandes locales. L'augmentation des performances provient non seulement de la réplication des UAL, mais aussi de l'indépendance de fonctionnement de celles-ci et de l'indépendance de traitement des paires d'opérandes. Un processeur vectoriel est donc constitué d'un certain nombre d'UAL, qui disposent chacune d'une mémoire (M), et qui sont associées à un module de contrôle central (CPU) (cf. figure 1.1).

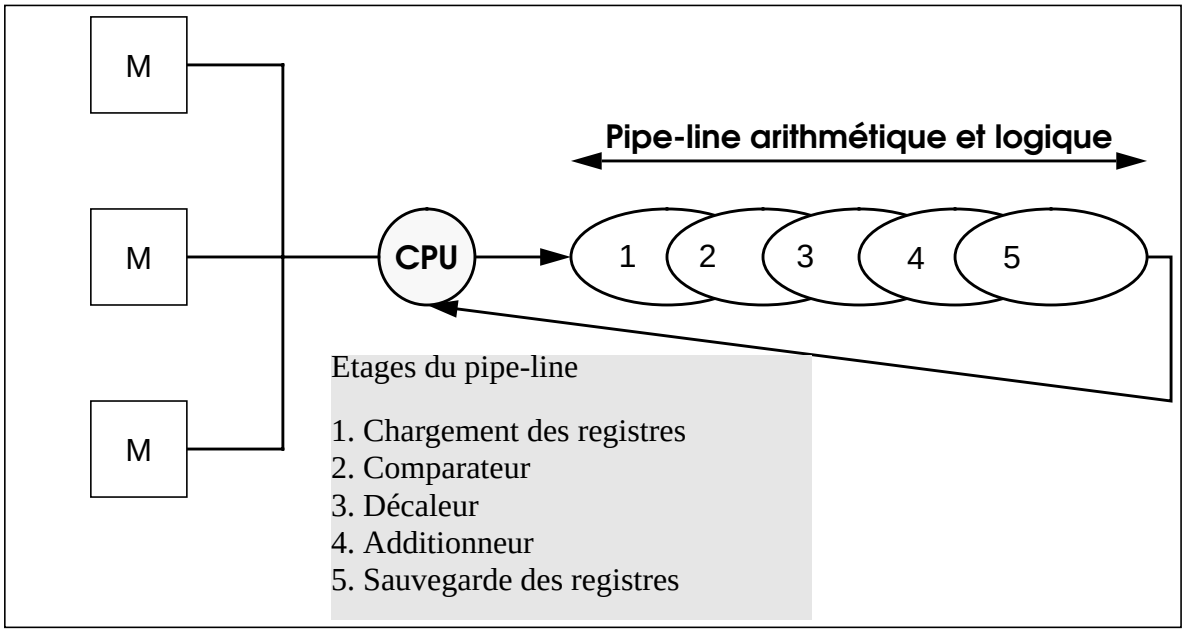


Figure 1.2 : architecture des processeurs pipe-line.

La technique du pipe-line consiste à faire se chevaucher l'exécution des différentes opérations. Pour cela les opérations sont décomposées en étapes élémentaires, communes et

1. On utilise également le terme anglais d'*array processor* (processeur en matrice) pour désigner les processeurs vectoriels, même si cet autre terme introduit la notion de multiplication des dimensions. On notera qu'il est aussi un synonyme de SIMD.

indépendantes; chaque étape correspondra à un étage du pipe-line (figure 1.2). Les opérations sont alors envoyées dans le pipe-line les unes après les autres pour y être exécutées étage par étage. Chaque étape étant indépendante des autres, lorsqu'un étage est affecté à une opération, l'opération suivante peut occuper l'étage précédent. Ainsi plusieurs opérations sont simultanément en cours d'exécution, et l'ensemble des circuits du processeur est plus efficacement exploité. Cette technique est employée pour produire des processeurs vectoriels qui minimisent le nombre de circuits et la consommation d'énergie, et sont par conséquent moins coûteux. Le pipe-line est aussi un facteur d'accroissement des performances lorsqu'il est utilisé dans les UAL des processeurs vectoriels, comme c'est le cas du **CRAY-1** [HocJes81, Kogge81] ou du **CDC Cyber 205** [Levine82].

Comparées aux processeurs vectoriels, les machines parallèles SIMD sont assez similaires: une même instruction est exécutée sur un ensemble d'unités de calcul. Ils diffèrent néanmoins dans leur approche: il ne s'agit plus de concevoir un processeur qui traite des données en parallèle, mais plutôt une machine composée de processeurs qui fonctionnent en parallèle. Ainsi ce qui est répliqué n'est plus l'unité de calcul arithmétique et logique mais un processeur, sous contrôle d'un séquenceur d'instructions central (SIC) (voir figure 1.3). De plus ces processeurs sont capables d'échanger des messages à travers un réseau d'interconnexion. Généralement régulier, ce réseau peut prendre diverses structures: en grille (l'ILLIAC IV [BKS68]), en hypercube (Connection Machine CM-2 [TucRob88, TMC90]).

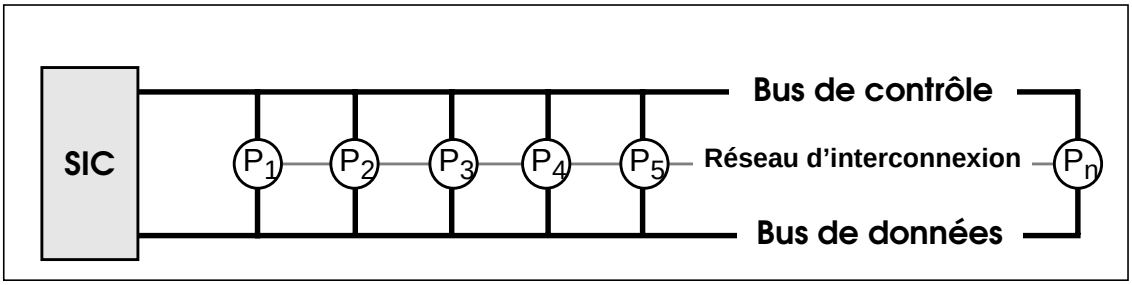


Figure 1.3 : architecture des machines SIMD.

1.1.2 Architectures MIMD à mémoire commune

Contrairement au fonctionnement synchrone des machines SIMD, dans les architectures MIMD l'indépendance des processeurs est totale: chaque processeur exécute son propre flot d'instructions indépendamment de ceux des autres; un programme n'est alors plus constitué d'une simple suite d'instructions, mais se compose de multiples flots de contrôle. Leur fonctionnement est donc **asynchrone**; le seul synchronisme qui peut exister est celui que les applications intègrent à leur code.

Pour réaliser des machines qui suivent ce mode de calcul plusieurs approches sont possibles. La plus simple consiste à dupliquer les unités de calcul autour de modules mémoire accessibles par tous. Là encore, pour partager la mémoire, il existe plusieurs solutions selon le type d'accès à la mémoire: accès uniforme (**UMA**[1]), accès non uniforme (**NUMA**[2]), accès non uniforme avec caches-mémoire (**COMA**[3]).

1. Uniform Memory Architecture.
2. NonUniform Memory Architecture.
3. Cache-Only Memory Architecture.

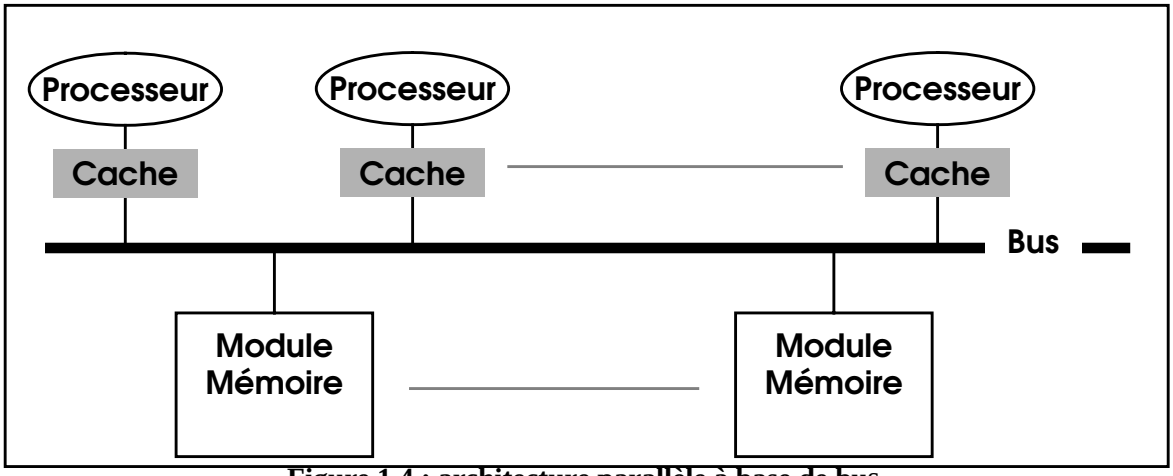


Figure 1.4 : architecture parallèle à base de bus.

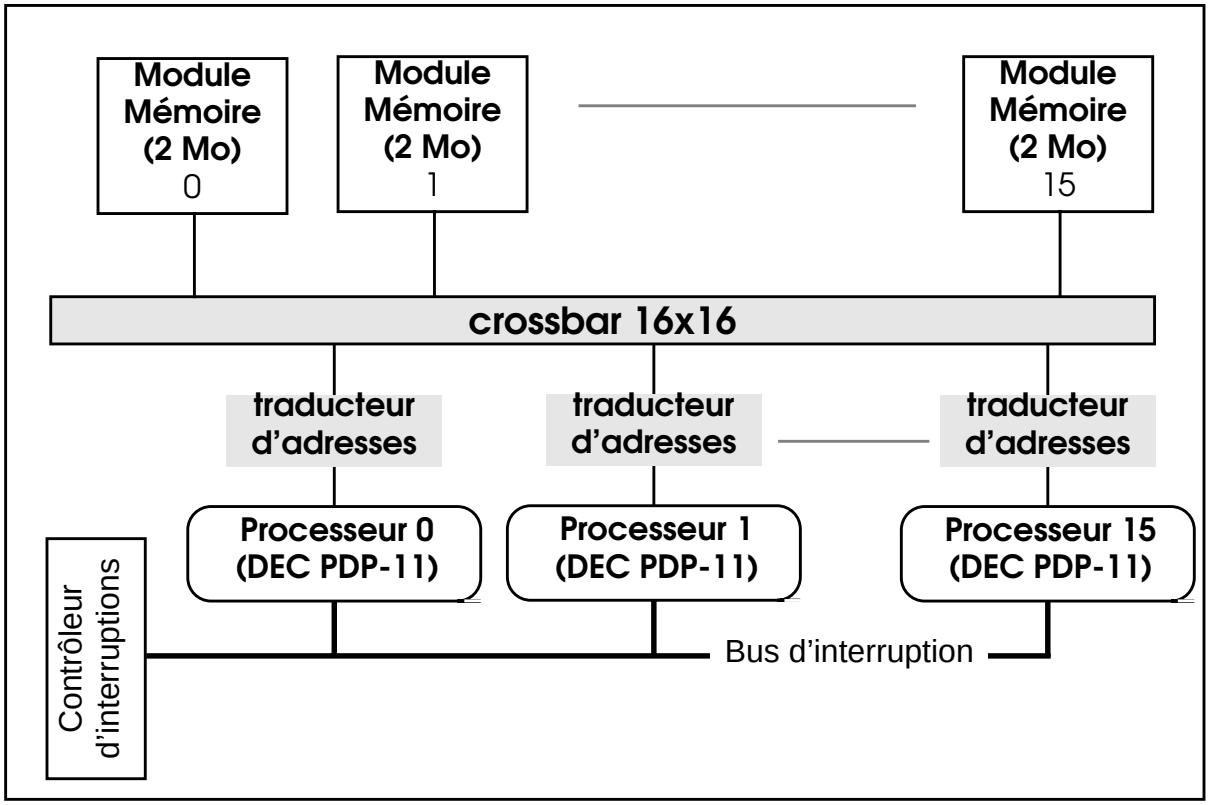


Figure 1.5 : le multi-processeur C.mmp.

Pour garantir un accès uniforme à la mémoire, les technologies utilisables sont celle du bus ou celle des réseaux de commutation non bloquants. Si la conception d'une machine parallèle à base de bus est plus aisée (figure 1.4), elle ne permet cependant pas d'atteindre un degré élevé de parallélisme; le partage du bus exige en effet un arbitrage et devient rapidement un goulot d'étranglement. Afin de limiter la contention au niveau du bus, des caches sont généralement utilisés; et leur cohérence est garantie par le matériel. Par contre, les réseaux de commutation non bloquants n'ont pas ce problème de congestion, les différents modules peuvent être accédés concurremment. C'est le cas par exemple du prototype **C.mmp** (Carnegie-Mellon multi-mini-processor) [WulBel72] dans lequel le réseau est un crossbar 16x16 (voir figure 1.5). Dans ces machines, dites fortement couplées, qu'elles soient construites avec l'une ou l'autre de ces techniques, le degré de parallélisme reste faible: le nombre maximal de processeurs est de l'ordre de la centaine.

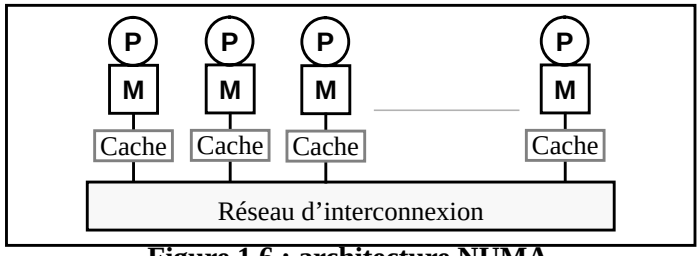


Figure 1.6 : architecture NUMA.

Pour dépasser ces contraintes technologiques, l'interconnexion complète des processeurs et des mémoires n'étant plus possible, dans les architectures NUMA le réseau de commutation non bloquant est remplacé par un réseau d'interconnexion, et la mémoire partagée est répartie entre les processeurs (figure 1.6). De ce fait les temps accès aux différents modules de mémoire ne sont plus constants, et varient en fonction des distances entre processeurs et mémoires. Afin de faciliter leur programmation, la plupart des machines NUMA intègrent des caches et un protocole de cohérence. Citons entre autres la **BBN Butterfly** qui utilise un réseau omega mais ne dispose pas de cache [RetTho86,CGRT85], et la **Stanford Dash** [Lenoski92]. Pour cette dernière les processeurs sont d'abord regroupés autour d'un bus, puis les groupes ainsi formés sont interconnectés par 2 grilles: une pour les requêtes et l'autre pour les réponses. Chacun des processeurs dispose d'un cache dont la cohérence est assuré par un protocole à base de répertoires, un pour chaque groupe de processeurs.

Quoique les multiprocesseurs à architecture COMA soient proches des machines NUMA avec caches, au sens où à tout instant chaque processeur dispose localement d'une partie de l'espace d'adressage global, la gestion de la mémoire, locale et globale, est ici plus évoluée. En effet la mémoire globale n'est plus statiquement distribuée, et les mémoires locales sont considérées, dans leur totalité, comme des caches secondaires. En plus d'être des caches, les mémoires locales peuvent aussi contenir des portions de l'espace d'adressage virtuel, et donc des données qui ne sont pas accédées localement. Ce mécanisme, qui porte le nom de «*ALLCACHE memory*» dans l'architecture de la **KSR1** [Rothnie92], est réalisé par un circuit spécialisé, le circuit *MAGIC* dans la machine **FLASH** de Stanford [Kuskin94].

1.1.3 Architectures MIMD à échange de messages

Comme nous venons de le voir pour les ordinateurs SIMD ou MIMD à mémoire commune, un haut degré de parallélisme n'est envisageable que si les unités de calcul sont faiblement couplées, et interconnectées par un réseau de topologie quelconque. C'est-à-dire qu'échanger des messages est incontournable dans les machines massivement parallèles. C'est aussi et surtout le paradigme de base pour la conception du support matériel des machines MIMD sans mémoire commune. Dans ce cas, le réseau d'interconnexion est l'élément principal de l'architecture d'une machine de ce type; son choix est donc un critère déterminant pour les applications.

Tout comme pour les calculateurs SIMD, les concepteurs et constructeurs de machines ont adoptés différents réseaux pour trouver un bon compromis entre connectivité et extensibilité[1], entre le diamètre du réseau et sa capacité à croître avec les mêmes contraintes. Dans ce contexte, trois structures ont suscité un vif intérêt: l'arbre, la grille, le tore, et l'hypercube.

1. capacité d'une machine à évoluer en taille.

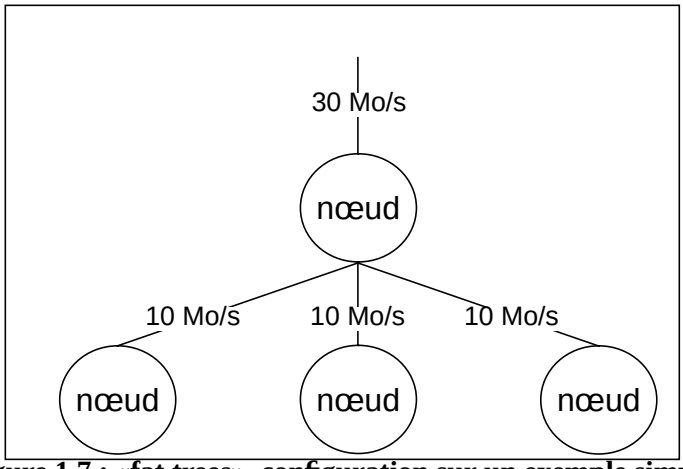


Figure 1.7 : «fat trees», configuration sur un exemple simple.

Un arbre, quel que soit le degré des noeuds (processeur + mémoire + interface de communication), est une structure naturellement évolutive. Cependant diamètre et congestion sont deux obstacles à l'utilisation de ces réseaux: le diamètre est en général élevé (le double de la profondeur de l'arbre) en raison des contraintes technologiques qui limitent le degré d'un nœud, et la congestion croît progressivement vers la racine qui est par conséquent un goulot d'étranglement. Pour éliminer ce phénomène de saturation les concepteurs de machines utilisent des arbres mieux adaptés: les «fat trees», dont la particularité est d'augmenter, en chaque nœud, la bande passante du lien de communication ascendant en proportion du nombre de descendants directs du nœud (voir figure 1.7); ainsi chaque lien dispose d'un débit suffisant pour éviter toute congestion. Toutefois cette solution est dépendante des possibilités des supports de communication et donc ne peut évoluer sans restriction. De plus pour compenser le diamètre important, les noeuds sont réduits à des circuits routeurs simples, spécialisés et rapides qui assument uniquement la tâche d'acheminement des messages. Feuilles de l'arbre, les processeurs et leurs mémoires disposent alors d'une bande passante constante et indépendante du diamètre.

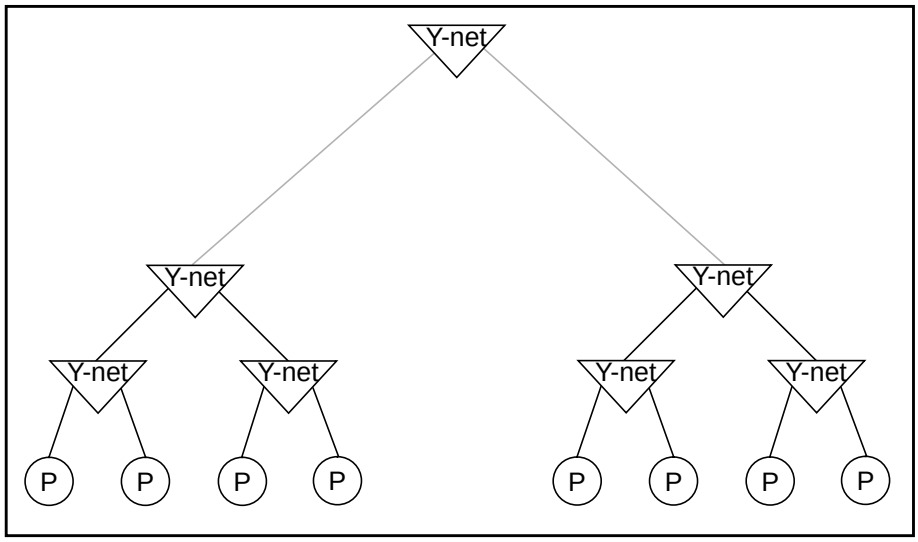


Figure 1.8 : architecture arborescente de la Teradata DBC/1012.

C'est cette technique qui a été choisie pour la **Teradata DBC/1012** (cf. figure 1.8) dans laquelle les routeurs (Y-net) sont capables d'effectuer une diffusion dans un sens et un tri dans l'autre [Tera84]. C'est également le cas pour les trois réseaux d'interconnexion (réseaux de

données[1], de contrôle[2] et de diagnostic[3]) de la **Connection Machine CM-5**, qui sont aussi basés sur les «fat trees» [Leis92].

Structure simple et proche de beaucoup d'applications (traitement d'images par exemple), la grille a été très usitée dans la conception des machines massivement parallèles. Son succès s'explique parce qu'elle est caractérisée par un diamètre raisonnable (égal à 2 fois la racine carrée du nombre de processeurs pour les grilles bidimensionelles carrées), et parce qu'elle n'exige qu'un degré fixe par nœud, indépendamment de la taille du réseau, et est donc en cela parfaitement extensible. La grille de la *Intel* **Paragon XP/S** [AlGot89] peut contenir plus de 4096 processeurs, celle du **Victor** d'*IBM* [Wilcke89] 256 processeurs auxquels peuvent être connectés plusieurs stations graphiques, ordinateurs frontaux, et processeurs d'entrées/sorties avec disques. La grille peut également être rebouclée en tore (cf. figure 1.9) pour réduire le diamètre, comme c'est le cas dans le **HSCP PAX** [Hosh86] dont l'autre particularité est de connecter chacun des 128 processeurs avec ses quatre voisins à partir d'une portion de mémoire commune, appelée *mémoire de communication*.

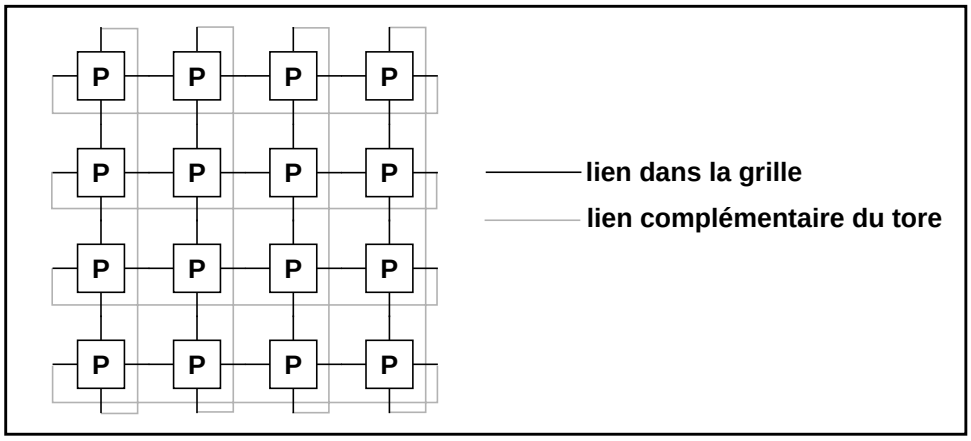


Figure 1.9 : un exemple de grille torique.

Très populaire également, l'hypercube offre un diamètre encore inférieur, égal au logarithme en base 2 du nombre de processeurs. Par contre l'extensibilité des machines construites à partir d'un hypercube est moins aisée qu'avec les structures précédentes. D'une part dans un hypercube le nombre de processeurs est toujours une puissance de 2, et il faut donc au moins doubler le nombre de processeurs pour augmenter la taille du réseau. D'autre part le degré n'est pas fixe et est égal à la dimension de l'hypercube, ce qui en fait une topologie plus difficilement extensible. Citons entre autres les *Intel* **iPSC** et **iPSC/860** [AlGot89] pour des hypercubes d'une dimension maximale de 7 (jusqu'à 128 processeurs), les **NCUBE/ten** et **NCUBE 2** [Hayes86] pour des dimensions 10 (1024 processeurs) et 13 (8192 processeurs).

La structure fixe et régulière de ces architectures les rend plus ou moins adaptées au regard de la structure logique des applications parallèles. Certes beaucoup d'applications ont tiré profit de l'un ou l'autre de ces réseaux, mais pour des problèmes non réguliers le placement des tâches peut s'avérer délicat et certaines parties du réseau saturées. Afin d'éviter ces dégradations et rendre efficace placement et équilibrage de charge, il faut en premier lieu prendre en compte la structure parallèle du problème, puis adapter au mieux le réseau à celle-ci.

1. qui assure la fonction de communication point-à-point.
2. pour les diffusions et combinaisons de données.
3. pour l'observation du fonctionnement des composants de la machine.

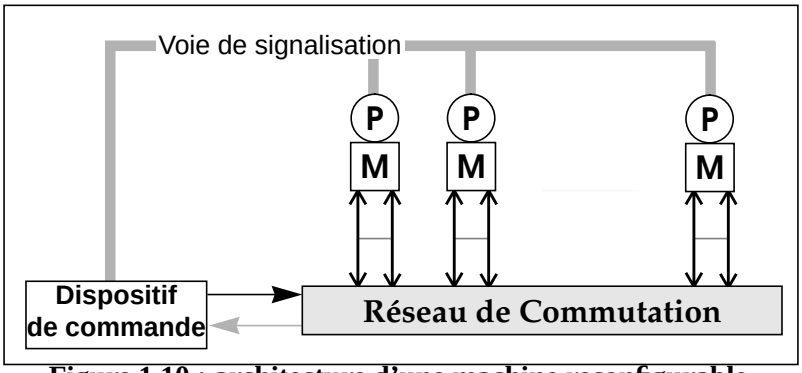


Figure 1.10 : architecture d'une machine reconfigurable.

C'est la voie choisie dans les **architectures reconfigurables** où il est possible de programmer la connectique du réseau d'interconnexion afin de trouver une configuration adéquate pour le problème à traiter. Ceci est réalisable aussi bien au cours de l'exécution de l'application qu'avant son chargement. En effet, au cours de l'évolution d'une application, une configuration statique du réseau, déterminée à l'avance, peut devenir inappropriée, et reprogrammer le réseau peut corriger cette situation. Dans ce cas, selon le type d'application, deux modes sont possibles: le mode synchrone qui nécessite la coordination de tous les noeuds de calcul, et le mode asynchrone d'utilisation délicate. La réalisation de telles machines requiert un réseau de commutation (réseau d'interconnexion programmable), un dispositif de commande, et éventuellement une voie de signalisation pour effectuer la coordination entre les unités de traitement (voir figure 1.10).

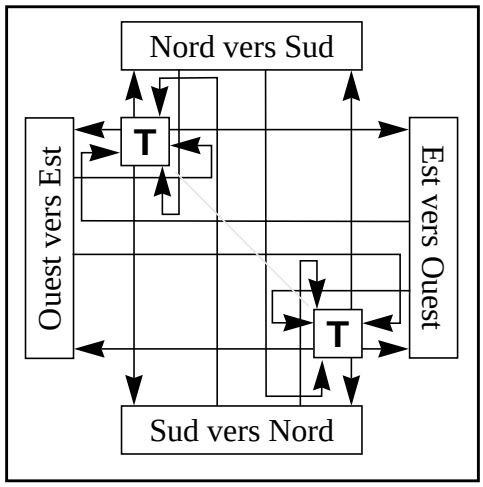


Figure 1.11 : les quatre réseaux du tandem Supernode.

L'architecture **Supernode**, qui est née d'une collaboration européenne à laquelle notre équipe a participée, illustre parfaitement cette classe d'ordinateurs [HJMW86, MunWai90, Waille91]. Modulaire, cette architecture est construite hiérarchiquement autour d'une configuration de base: le tandem, qui peut contenir 16, 32 ou 64 processeurs. Dans un tandem le réseau de commutation est en fait décomposé en quatre sous-réseaux (figure 1.11), chacun capable de connecter n entrées à n sorties à l'aide d'un crossbar; cette décomposition permet en effet de réduire au quart la taille du réseau. De plus, ceci s'adapte parfaitement au processeur choisi, le transputer (T), qui est doté de quatre liens de communication bi-directionnels, notés généralement *Nord*, *Sud*, *Est* et *Ouest*. Ici, chacun des sous-réseaux permet de réaliser des connexions *Nord* vers *Sud*, *Sud* vers *Nord*, *Est* vers *Ouest* et *Ouest* vers *Est*. La gestion de la configuration des quatre réseaux du tandem est assurée par deux processeurs supplémentaires: les *contrôleur*

(C). Ces contrôleurs dialoguent avec les processeurs par l'intermédiaire d'une voie de signalisation, appelée *voie de contrôle*.

Ce sont les contraintes technologiques des crossbars qui limitent le nombres de processeurs d'un tandem. Pour obtenir des machines de plus grande taille, de plus de 64 transputers, le nombre de processeurs des tandems est réduit de moitié (32 transputers) afin de libérer des connexions sur les quatre réseaux. Il est alors possible d'ajouter un deuxième niveau hiérarchique d'interconnexion à partir de crossbars. Ces deux niveaux de crossbars forment alors un réseau de Clos à trois étages (voir [Clos53]) entre l'ensemble des processeurs. La particularité est ici que les connexions intra-tandem se font uniquement à partir des commutateurs internes au tandem. L'ensemble de la machine est sous le contrôle d'un transputer additionnel: le *superviseur* (S). Enfin, la voie de contrôle est partagée entre tous les tandems ainsi que par le superviseur. La figure 1.12 schématise cette architecture.

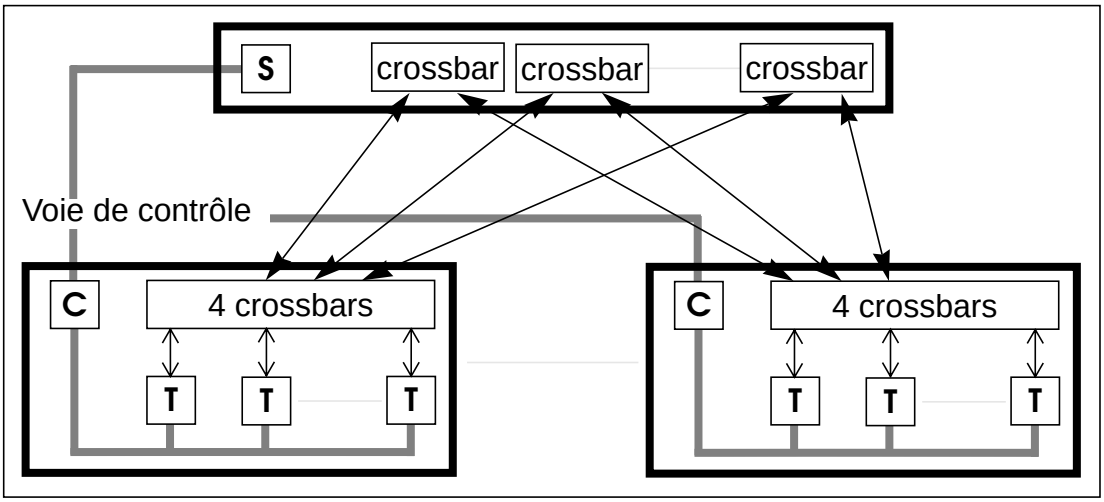


Figure 1.12 : structure hiérarchique des machines Supernode.

1.2 Problèmes liés au parallélisme

Du point de vue de l'utilisateur la difficulté majeure est de concevoir une application, de se détacher du modèle séquentiel pour décomposer son problème en tâches et de définir les interactions entre ces tâches. La solution la plus simple consiste à décrire le problème dans un langage séquentiel classique, puis à appliquer un «paralléliseur», c'est-à-dire un *système automatique de transformation de programmes*, pour obtenir un code exécutable par une machine parallèle. L'inconvénient dans ce cas est que l'utilisateur décrit un algorithme séquentiel, et par conséquent le code extrait par ce type d'outils ne peut exprimer une décomposition «efficace» de l'application. Naturellement avec le développement des diverses architectures, et c'est une alternative aux «paralléliseurs», sont apparus divers modèles de programmation parallèle: **PRAM**[1] [ForWyl78], **CSP**[2] [Hoare78], et les **Acteurs** ou **Actors** [Hewitt77, Agha86] en sont des exemples.

Mais ces solutions, sans doute parce qu'elles se situent à un niveau trop élevé, celui de l'utilisateur, ne répondent pas au problème de l'exploitation d'une machine parallèle. Pour cela, quelle que soit l'architecture considérée, il faut être capable d'utiliser au maximum chaque processeur. Pour un ordinateur SIMD, il s'agit d'arranger, d'*ordonnancer* les calculs pour utili-

1. Parallel Random Access Memory (mémoire à accès parallèles aléatoires).
2. Communication Sequential Processes (processus séquentiels communicants).

ser constamment le plus grand nombre de processeurs et/ou pour remplir au mieux le pipe-line. Ordonnancer efficacement des calculs n'est pas spécifique au parallélisme: c'est aussi une tâche importante de tout système multi-programmé. Ainsi, les solutions apportées dans le cas du parallélisme sont proches et découlent de celles utilisées dans les systèmes multi-programmés. Par contre avec une architecture MIMD, il faut en plus opérer une *allocation*, la plus optimale possible, des tâches (calculs) aux processeurs, et le cas échéant effectuer un *équilibrage de la charge* des processeurs. El-Ghazali Talbi et Ahmed Elleuch, au sein de notre équipe de recherche et au cours de leurs thèses respectives [Talbi93, Elleu94], ont étudié ces problèmes et y ont apporté des solutions originales

Enfin si l'on considère les machines MIMD à échange de messages, il est inévitable de *contrôler les communications*. C'est aux problèmes liés au contrôle des communications, et plus particulièrement ceux posés par les communications globales que cette thèse est consacrée.

1.3 Contrôle des communications

La recherche dans le domaine de la communication est ancienne: elle a débuté avec la téléphonie, puis s'est développée avec les réseaux d'ordinateurs. Aujourd'hui ces activités persistent, que ce soit pour prospecter de nouveaux média ou pour concevoir de nouvelles méthodes de communication; mais leur application dans la conception des machines parallèles a ouvert une nouvelle voie de recherche dans ce domaine. Ce sont surtout les résultats obtenus dans les réseaux de grande taille (**WAN**[1]), ou réseaux de topologies connexes quelconques, qui ont le plus inspiré les concepteurs de machines massivement parallèles. Cependant l'état des connaissances sur ce type de réseaux reste insuffisant pour complètement maîtriser les communications dans de tels ordinateurs. Or le réseau est ici un élément fondamental, et son contrôle crucial pour garantir une utilisation correcte, mais aussi la plus efficace possible, des ressources disponibles. Il s'agit donc là d'un axe de recherche des plus importants du parallélisme.

Echanger des messages – ou plus exactement router – à travers un réseau général met en oeuvre une technique d'*acheminement* et une *fonction de routage*. La fonction de routage détermine le chemin que doit emprunter chaque message pour arriver à destination, et la technique d'acheminement définit comment doivent être utilisées les ressources du réseau tout au long du chemin qui lui a été attribué. La difficulté de l'acheminement réside dans une utilisation efficace des ressources allouées pour communiquer: il faut éviter de réserver une ressource en la laissant inexploitée durant un temps trop long pour ne pas réduire la bande passante du réseau et, peut-être retarder inutilement des messages. Mais il n'est pas non plus recommandé d'introduire des délais supplémentaires en chaque noeud alors que les ressources nécessaires sont libres.

La *correction* est la première exigence que doit remplir toute fonction de routage. Pour cela toute communication doit préserver les trois propriétés suivantes: absence d'*interblocage*, absence de *famine*, *validité*. L'interblocage se produit lorsque se forme dans le réseau un cycle d'attente de ressources. Il y a famine lorsqu'un message reste indéfiniment bloqué en attente d'une ressource qui se libère mais est toujours attribué à un autre message; la cause de cette situation est généralement le manque d'équité entre les messages. La validité est violée si un message n'atteint jamais sa destination, soit parce que la fonction de routage ne définit pas un chemin entre toute paire de processeurs dans un routage déterministe, soit parce que le système de routage dispose de plusieurs chemins et oriente le message sur des routes toujours différentes, pour des raisons d'adaptabilité par exemple. Bien que la correction soit vitale, elle n'est

1. Wide Area Network.

néanmoins pas suffisante vis-à-vis des contraintes qu'impose le parallélisme massif. Il faut de surcroît optimiser, et les chemins qui sont induits, et leur représentation interne: minimiser les chemins pour réduire la latence et encombrer le moins possible le réseau, et rester indépendant de la taille du réseau dans la quantité de mémoire nécessaire à la fonction de routage.

Acheminer un message d'un processeur à un autre n'est donc pas chose facile, et cela peut s'avérer très complexe si l'on veut tenir compte du comportement dynamique d'un réseau au cours du temps, ou si l'on veut pouvoir réaliser, à ce niveau, des schémas d'échanges plus élaborés tels que la *diffusion* d'un même message à plusieurs processeurs, ou la réduction d'un message de plusieurs sources vers un unique récepteur. Prendre en compte l'évolution du réseau, les congestions momentanées de certaines régions, et appliquer un *routage adaptatif* ne peut qu'améliorer les performances; ce qui correspond bien à nos objectifs. De même rendre le routage capable de diffuser, ou plus généralement d'effectuer des communications globales, permet aussi d'éviter de surcharger le réseau en évitant qu'à un niveau supérieur cela se traduise par l'envoi d'un message séparé pour chaque processeur concerné.

Du point de vue de l'application le routage de messages est insuffisant, simplement parce que les processus de celle-ci ont recours à un mode de communication plus sophistiqué que le simple envoi de messages vers un processeur destinataire, ou même plusieurs. En premier lieu un processus ne communique qu'avec d'autres processus et non avec des processeurs, il faut donc localiser les processus destinataires des divers messages, puis sur le site de réception distribuer les messages selon leur(s) destinataire(s). En outre lorsque la technique d'acheminement impose une taille de message fixe, il est nécessaire de découper les messages en paquets, puis il faut les réassembler. Par ailleurs pour leur portabilité les applications doivent être aussi indépendantes de l'architecture que possible. Enfin et surtout les interactions mises en oeuvre astreignent les processus à respecter des disciplines de communication spécifiques. Construits au-dessus du routage les *protocoles* accomplissent ces tâches. Ils sont eux aussi confrontés aux problèmes d'efficacité, d'interblocage, de famine et de correction.

1.4 Modèles de communication basés sur la diffusion

Parmi les problèmes de communication, la diffusion, et plus largement les communications globales, sont avant tout un moyen pour faire communiquer, se synchroniser et coopérer les multiples composantes d'une application parallèle. Même s'il est toujours possible, aux utilisateurs chevronnés, de reconstruire une telle fonctionnalité en y apportant des modifications qui puissent la rendre plus spécifiquement adaptée à l'application, il est de toute façon bien plus profitable d'utiliser une primitive du système de communication sous-jacent prévue à cet effet, tant pour la portabilité que pour l'efficacité.

Selon ce principe des modèles de programmation fondés sur la diffusion ont été proposés: ***BSP***[1] qui est issu de ***CSP*** [Hoare78], et ***CBS***[2] construit sur le modèle ***CCS***[3] [Mil89]. Ce sont en fait les seuls modèles théoriques, à notre connaissance, qui permettent la spécification et la validation de systèmes parallèles à diffusion.

1. Broadcasting Sequential Processes: processus communiquant par diffusions.
2. Calculus of Broadcasting Systems: calcul pour systèmes à diffusion.
3. Calculus of Communicating Systems: calcul pour systèmes communicants.

1.4.1 BSP

BSP [Geh84] est le premier exemple de modèle de programmation basé sur la diffusion. Il est clairement dérivé du modèle synchrone CSP dont il reprend d'ailleurs la sémantique de communication point-à-point. Nous retrouvons les opérateurs d'émission: **!**, et de réception: **?**; l'opérateur ? a été toutefois modifié pour pouvoir exprimer la réception d'un message en provenance d'un processus quelconque, avec une priorité *premier entré/premier sorti*: **any**?m.

Cependant, contrairement à CSP où tous les échanges de messages se font lors de rendez-vous, les diffusions dans BSP sont asynchrones. BSP a été défini pour les architectures de réseaux locaux où l'attente de réception n'a pas été considérée par N. H. Gehani comme réaliste [Geh84]. Il modélise donc des processus «diffusants» qui ne conditionnent pas leurs communications par rapport à l'existence de récepteurs. Pour un émetteur, les processus qui ne sont pas prêts à recevoir perdront son message; et c'est au programmeur de prendre en compte ce comportement dans la conception de ses programmes.

Lorsque l'émetteur, après avoir diffusé un message doit attendre une ou plusieurs réponses, un mécanisme d'alarme doit donc être introduit, parce que le processus ne peut déterminer a priori le temps d'attente de toutes les réponses. De plus, comme il n'y a aucune garantie que le message soit effectivement reçu par des processus récepteurs, cela peut conduire l'émetteur à rediffuser son message ou à déclencher une action alternative. Afin de pouvoir exprimer de tels comportements, BSP comprend un opérateur d'attente: **delay(**temps**)**.

Dans le modèle tout message *m* est une association entre l'identification de l'émetteur *m.sender*, et les informations *m.info* véhiculées. Le champ *m.info* est fourni par l'émetteur alors que *m.sender* est géré par le système de communication. En réception, la sélection d'un message peut se faire sur les valeurs de ces champs.

BSP possède une syntaxe proche de celle de CSP, avec les commandes gardées de Dijkstra [Dijk75]. Il offre deux opérateurs de diffusion: ○ et ⊗. Tous deux sont asynchrones et ne diffèrent que dans la manière dont les messages doivent être gérés par le récepteur: le premier ne requiert aucun espace tampon pour stocker les messages, alors que pour le second un espace tampon de taille infinie doit être associé à tout processus. Il s'agit donc de diffusions dans les deux situations extrêmes: pour le premier opérateur un message diffusé n'est réceptionnable que par les processus prêts à le recevoir, et dans le cas opposé aucun message ne peut être perdu par les récepteurs. Ces deux opérateurs permettent, l'un et l'autre, la diffusion totale vers tous les processus, et la diffusion sélective vers un sous-ensemble de processus:

- **all**○m: le message *m* est envoyé à tous les processus prêts à recevoir;
- **all**⊗m: le message *m* est envoyé dans le tampon de tous les processus;
- **[**p_1**,** p_2**, ..,** p_n**]**○m: *m* est envoyé aux processus prêts à recevoir parmi l'ensemble $\{p_1, p_2, ..., p_n\}$;
- **[**p_1**,** p_2**, ..,** p_n**]**⊗m: *m* est envoyé dans le tampon des processus de l'ensemble $\{p_1, p_2, ..., p_n\}$.

Quel que soit le mode d'émission des messages, BSP ne définit qu'un seul opérateur de réception: ⍰. Cet opérateur permet la sélection d'un message selon son expéditeur. Il s'agit d'un opérateur bloquant qui interrompt l'exécution du processus appelant tant qu'aucun message ne peut être reçu. Par ailleurs il est également possible d'appliquer la sélection par rapport au contenu du message, et même de mélanger les deux critères:

- **any**?m: réception d'un message *m* depuis une source quelconque et de n'importe quel contenu;
- **any(sender=[**p_1, p_2, ..., p_n**])**?m: réception d'un message *m* depuis une source de l'ensemble $\{p_1, p_2, ..., p_n\}$, et de contenu indifférent;
- **any(info=**i**)**?m: réception d'un message *m* depuis une source quelconque mais de contenu *i*;
- **any(info=**i**,sender=[**p_1, p_2, ..., p_n**])**?m: réception d'un message *m* depuis une source de l'ensemble $\{p_1, p_2, ..., p_n\}$, et de contenu *i*.

Outre l'introduction d'opérateurs asynchrones dans un modèle original synchrone, nous retiendrons de BSP ces deux modes de diffusion: l'un sans espace tampon qui correspond à un protocole simplifié, et le second avec un espace tampon infini dont l'implémentation n'est pas toujours efficace.

1.4.2 CBS

L'objectif de CBS [Pra91, Pra93] est de définir un modèle théorique qui représente le mieux possible les architectures matérielles à base d'un médium à diffusion, et plus précisément les réseaux locaux. La diffusion est selon ce principe une opération «audible» par tout composant d'un système, mais perçue seulement par ceux qui sont à l'écoute du médium. Les communications dans CBS sont donc de type «radiophonique», et en cela ce modèle, quoique plus théorique et formel, se rapproche de BSP. Cependant, alors qu'en BSP la diffusion est totalement asynchrone et dispose d'un mode de diffusion avec tampons infinis, CBS ne considère que le mode sans tampons où tout message a un seul transmetteur et zéro ou plusieurs récepteurs: ceux qui captent immédiatement le message. Bien que le synchronisme ne soit pas réellement exprimé, cette notion d'«isochronie» montre clairement la simultanéité dans le temps entre une émission et les réceptions.

Les ***systèmes*** que permet de modéliser CBS sont composés d'***agent***s dont l'identité persiste dans le temps, et dont le comportement consiste en une suite d'***actions*** de communications: émission ou réception. Ces agents évoluent d'***état*** en ***état*** par les actions qu'ils effectuent. Il n'y a pas de distinction formelle entre un agent, un état et un système.

L'élément de communication entre ces agents est le ***message***, il transporte deux informations: un en-tête sous la forme d'un ***nom*** ou ***signal*** qui spécifie qui peut lire le message, et une ***valeur*** qui est son contenu. Un message de nom *a* et de valeur *v* est dénoté *a(v)*.

Tout message est reçu par l'ensemble des systèmes connectés au réseau, et peut être lu par tout sous-ensemble de ceux-ci. Pour recevoir des messages les récepteurs se mettent en observation des en-têtes et éliminent les messages qui ne correspondent pas au filtre que l'agent s'est fixé. Usuellement, seulement un agent à la fois peut émettre un message, et toute collision est détectée, supprimée et réparée selon des algorithmes employés dans les réseaux locaux.

Un agent, qui peut être défini par rapport à d'autres agents, est une composition d'actions. La spécification de son comportement est réalisée avec pour chaque action un préfixe suivi de l'expression du comportement de l'agent après l'action. Les deux actions de base sont la transmission dont le préfixe est *a(v)!*, et l'observation des messages dont le préfixe est *a(x)?*[1].

1. filtrage des messages de nom *a*.

Comme premier exemple, l'agent *a(v)!P* ne peut effectuer qu'une seule action : la transmission de *a(v)* ; après quoi il deviendra *P*. Ceci s'écrit également, à l'aide des systèmes de transitions étiquetés :

$$a(v)!P \xrightarrow{a(v)!} P$$

De même l'agent *a(x)?Q(x)*, à la lecture de *a(v)*, se comporte comme *Q(v)* ; ce qui s'écrit :

$$a(x)?Q(x) \xrightarrow{a(v)?} Q(v)$$

Un agent particulier o est défini pour la spécification d'aucune action à réaliser, que ce soit l'observation ou la transmission ; il marque l'état final d'un agent. Le comportement de ces agents vis-à-vis d'autres messages est clair : il conduit à leur élimination. C'est-à-dire que *a!P* ne peut que transmettre *a*, et ignorer tout autre message, *a(x)?Q(x)* fait de même à l'égard de tout message dont le signal n'est pas *a*, et o élimine tout message. Ce filtrage se traduit, dans le diagramme d'états, par une transition de l'état courant vers lui-même. Pour chacun des messages *a* éliminés, cela se traduit en fait par un prédicat «ne pas observer *a*», qui s'écrit *a(v):*, et qui a l'effet suivant sur un agent *F* :

$$F \xrightarrow{a(v):} F$$

Le parallélisme des agents est spécifié par un opérateur dit de composition parallèle : $|$, où $E \mid F$ signifie l'exécution simultanée des agents *E* et *F*. La composition parallèle d'agents suit l'ensemble des règles du tableau 1.1, pour la communication, les lectures conjointes, le recouvrement d'actions, la relation entre une lecture d'un message par un agent et son élimination par un autre agent, et les éliminations conjointes.

communication	$\dfrac{E \xrightarrow{a(v)!} E' \quad F \xrightarrow{a(v)?} F'}{E \mid F \xrightarrow{a(v)!} E' \mid F'}$	règle commutative
lectures conjointes	$\dfrac{E \xrightarrow{a(v)?} E' \quad F \xrightarrow{a(v)?} F'}{E \mid F \xrightarrow{a(v)?} E' \mid F'}$	
recouvrement	$\dfrac{E \xrightarrow{a(v)!} E' \quad F \xrightarrow{a(v):} F}{E \mid F \xrightarrow{a(v)!} E' \mid F}$	règle commutative
lecture/élimination	$\dfrac{E \xrightarrow{a(v)?} E' \quad F \xrightarrow{a(v):} F}{E \mid F \xrightarrow{a(v)?} E' \mid F}$	règle commutative
éliminations conjointes	$\dfrac{E \xrightarrow{a(v):} E \quad F \xrightarrow{a(v):} F}{E \mid F \xrightarrow{a(v):} E \mid F}$	

Tableau 1.1 : règles de composition parallèle dans CBS.

Le produit des différentes actions d'agents exécutés en parallèle, loi o, n'est définie qu'entre des actions qui ont trait au même message, puisqu'il ne peut y en avoir qu'un et un seul à tout instant. Le produit de transmissions est indéfini, alors que tout autre produit est le résultat de l'action de plus haut rang, en considérant la transmission (*a(v)!*) en premier, puis l'observation (*a(v)?*) et enfin le rejet (*a(v):*). Il s'agit d'une opération commutative et associative.

L'opération de somme est comme en CCS disponible pour les émissions et lectures, mais pas pour les éliminations. $a?P+b!Q$ est une telle somme et signifie : émettre b et devenir Q, à moins qu'un message a puisse être reçu et dans ce cas se comporter comme P. S'il y a réception d'un message c, alors les deux alternatives le supprime.

La récursion, pour l'élimination, exprime syntaxiquement que le rejet de messages ne cause aucun changement d'état ; d'où :

$$recX.b?X \xrightarrow{a:} recX.b?X$$

Afin de restreindre les champs d'écoute et de communication d'un sous-système à un ensemble réduit de signaux, CBS définit deux opérateurs unaires : la «sourdine» et le renommage. Le renommage, ϕ, permet simplement de traduire le nom d'un message, d'un sous-système où il garde son nom d'origine, vers les autres sous-systèmes. Dans l'exemple suivant nous avons deux agents F et E qui «parlent» des langues différentes : f pour F (français) et a pour A (anglais), dans le système global I (international) :

$$I \stackrel{def}{=} E[\phi] \mid F[\phi]$$
$$A \stackrel{def}{=} a!A$$
$$\mathrm{F} \stackrel{def}{=} f!F$$

Au sein de F tout message émis garde son nom f, et est transmis au sous-système A avec une traduction du nom : $\phi(f!) = a!$. Il en est de même pour A vis-à-vis de F où $\phi(a!) = f!$.

La sourdine permet de confiner certains messages dans un sous-système. Il agit d'un ensemble de signaux pour lesquels le sous-système donne une signification privée, non reconnue à l'extérieur. Pour l'agent $P\backslash\{a\}$ toutes les émissions de $a(v)$ sont visibles par les sous-agents de P, mais invisibles à l'extérieur. Toutefois il s'agit d'une communication, et par conséquent tous les agents doivent recevoir un message. A l'extérieur $a(v)!$ sera remplacé par une action silencieuse $\tau!$ qu'aucun agent ne peut effectivement réceptionner.

Soient S l'ensemble des noms de messages (signaux), $\tau \notin S$ le signal silencieux, V l'ensemble des valeurs de messages et $S_V = S \times V$. L'ensemble des messages est donc $S_V \cup \{\tau\}$, l'ensemble des actions d'émission et de réception $\mathrm{Act} = (S_V \times \{?,!\}) \cup \{\tau!\}$ et l'ensemble total des actions $Act^{:} = Act \cup (S_V \times \{:\}) \cup \{\tau:\}$. Nous pouvons alors donner la syntaxe d'une expression CBS :

$$E ::= X \mid o \mid \tau!E \mid a(v)!E \mid a(x)?E \mid E+E \mid E \mid E \mid E\backslash\mathcal{N} \mid E[\phi] \mid recX.E$$

où X est une variable, $\mathcal{N} \subseteq S$ un ensemble de signaux, x une variable interne à un agent, $a \in S$ un signal et $v \in V$ une valeur de message.

Le lecteur intéressé par la sémantique des opérateurs de CBS peut se reporter au tableau 1.2. Dans ce tableau $(a,b) \in S^2$ avec $a \neq b$, $(u,v) \in V^2$ avec $u \neq v$, $\mu \in Act$, $(\upsilon, \upsilon_1, \upsilon_2) \in Act^{:2}$, et $\mathcal{N} \subseteq S$.

opérateur	transmission	observation	élimination
o			$o \xrightarrow{a(v):} o$
Expression			$E \xrightarrow{\tau:} E$
!	$\tau!E \xrightarrow{\tau!} E$ $a(v)!E \xrightarrow{a(v)!} E$		$\tau!E \xrightarrow{a(v):} \tau!E$ $a(v)!E \xrightarrow{b(u):} a(v)!E$
?		$a(x)?E(x) \xrightarrow{a(v)!} E(v)$	
+	$\dfrac{E \xrightarrow{\mu} E'}{E+F \xrightarrow{\mu} E'}$	$\dfrac{E \xrightarrow{\mu} E'}{F+E \xrightarrow{\mu} E'}$	$\dfrac{E \xrightarrow{a(v):} E \quad F \xrightarrow{a(v):} F}{E+F \xrightarrow{a(v):} E+F}$
|	o | $a(v)!$ $a(v)?$ $a(v):$ $a(v)!$ | $a(v)!$ $a(v)!$ $a(v)?$ | $a(v)!$ $a(v)?$ $a(v)?$ $a(v):$ | $a(v)!$ $a(v)?$ $a(v):$ o | $\tau!$ $\tau:$ $\tau!$ | $\tau!$ $\tau:$ | $\tau!$ $\tau:$ $\dfrac{E \xrightarrow{\upsilon_1} E' \quad F \xrightarrow{\upsilon_2} F'}{E \mid F \xrightarrow{\upsilon_1 o \upsilon_2} E' \mid F'}$		$\upsilon_1 o \upsilon_2$ défini si, et seulement si, les messages portent le même nom et ont la même valeur
\	$\dfrac{E \xrightarrow{\upsilon} E'}{E\backslash\mathcal{N} \xrightarrow{\upsilon} E'\backslash\mathcal{N}}$ avec le nom de υ n'appartenant pas à $\mathcal{N}$ $\dfrac{E \xrightarrow{a(v)!} E'}{E\backslash\mathcal{N} \xrightarrow{\tau!} E'\backslash\mathcal{N}}$ $a \in \mathcal{N}$		$E\backslash\mathcal{N} \xrightarrow{a(v)!} E\backslash\mathcal{N}$, $a \in \mathcal{N}$
ϕ	$\dfrac{E \xrightarrow{\upsilon} E'}{E[\phi] \xrightarrow{\phi(\upsilon)} E'[\phi]}$		
rec	$\dfrac{E[recX.E/X] \xrightarrow{\upsilon} E'}{recX.E \xrightarrow{\upsilon} E'}$		$\dfrac{E[recX.E/X] \xrightarrow{a:} E[recX.E/X]}{recX.E \xrightarrow{a:} recX.E}$

Tableau 1.2 : sémantique de CBS.

1.5 Sujet de cette thèse

Parmi les différentes approches possibles et parmi les divers outils qu'il est possible d'offrir, nous avons choisi d'aborder le parallélisme du point de vue du système d'exploitation. Encore aujourd'hui l'accès à ce genre de machines se fait trop souvent à partir d'un frontal, et peu d'entre elles disposent d'un réel système d'exploitation autonome et adapté. Dans bien des cas le contrôle d'accès se fait dans un mode pour le moins éculé: le traitement par lots, où les utilisateurs, à tour de rôle, disposent exclusivement de la machine pour exécuter une de leurs applications. Certes, avec certains modèles il est possible de définir des partitions et ainsi de partager la machine, mais cette possibilité est limitée et ne répond pas complètement aux attentes des utilisateurs, depuis longtemps habitués aux systèmes multiprogrammés et au temps partagé. Résorber ce handicap est à nos yeux une motivation scientifique importante, et c'est l'objectif de la plus grande part des recherches de notre équipe. Nous nous plaçons en fait dans le cadre des systèmes à base de micro-noyau, avec toutefois des particularités propres au parallélisme.

Bien que notre but soit d'obtenir un système d'exploitation général et disponible pour le plus grand nombre de machines, nous n'avons cependant pas considéré les calculateurs SIMD d'un fonctionnement trop particulier. Seules les architectures MIMD ont été prises en compte, et plus spécifiquement les architectures à échange de messages parce qu'elles posent des problèmes ardus pour l'exploitation efficace du parallélisme massif. Cette thèse est tout particulièrement destinée à cette catégorie d'ordinateurs.

Dans ce cadre cette thèse est consacrée au contrôle des communications, mais il s'agit ici de s'intéresser surtout aux diffusions et de ne plus considérer que seulement les échanges point-à-point. Nous apportons ici une solution nouvelle et générale au problème du routage pour la diffusion; non seulement notre algorithme de diffusion est capable d'acheminer un même message vers plusieurs destinations, mais il est aussi indépendant de la topologie du réseau d'interconnexion, correct, et suffisamment simple pour être intégré dans un circuit routeur. Il est important de noter que nos motivations dans ce domaine ne sont pas d'effectuer une étude théorique sur des solutions possibles pour des architectures ciblées, mais bien d'apporter une solution nouvelle à la diffusion pour une classe d'architectures la plus vaste possible, et de réaliser cette solution.

L'intense activité de recherche [CNL89] menée au cours des dernières années, conjugée avec la définition de nouveaux modèles de communication fondés sur la diffusion, montre s'il en était besoin, l'intérêt des mécanismes de diffusion. Quoique bien plus coûteuse qu'une communication point-à-point, elle peut s'avérer plus efficace et plus simple que tout autre mécanisme dans un grand nombre d'algorithmes tels que ceux employés en synthèse d'images, pour la simulation de circuits, ou dans la résolution d'un grand nombre de problèmes. Par exemple, son utilisation dans la résolution de systèmes d'équations, par la méthode du pivot de Gauss, semble naturelle: à chaque étape le pivot sélectionné est simplement diffusé.

Enfin dans le cadre de nos travaux et réalisations, au sein de notre système d'exploitation PAROS/ParX [BFF89, BCD93], nous proposons une implantation logicielle d'un service de diffusion utilisable au niveau de l'application, i. e. pour les processus qui la composent. Ce service se compose de deux protocoles qui sont construits au-dessus de notre routeur à diffusion; ils sont suffisamment généraux pour envisager l'implémentation des différents modèles, langages et interfaces qui en font usage.

1.6 Plan du manuscrit

Ce rapport est divisé en trois parties : tout d'abord une étude de la place de la diffusion dans le parallélisme, puis la présentation de notre routeur à diffusion, et enfin la définition d'une machine virtuelle à diffusion: ***PDVM***[1]. La suite de cette première partie est consacrée à la diffusion et aux communications globales dans les langages, systèmes d'exploitation et environnements de développement et d'exécution adaptés au parallélisme.

Le second volet de ce rapport aborde la question du routage des messages dans les architectures massivement parallèles, et plus spécifiquement du routage pour la diffusion. Nous nous intéresserons au routage point-à-point dont nous donnerons les principales solutions, et plus particulièrement les deux fonctions de routage conçues par les membres de notre équipe. Ceci fait nous étudierons et critiquerons les différentes solutions qui ont été proposées pour résoudre le problème de la diffusion. Nous terminerons par la présentation de notre routeur, dont nous apporterons la preuve de sa correction. Nous montrerons également que notre routeur est indépendant de la topologie du réseau, et comment il est possible de l'intégrer dans un circuit.

La dernière partie sera consacrée à la diffusion inter-processus. Face aux diverses formes de diffusions observées dans cette thèse, nous dégagerons l'architecture d'une machine virtuelle pour le système PAROS/ParX, ainsi que la sémantique que doivent respecter les protocoles qui la constitue.

Enfin nous conclurons sur l'ensemble des résultats que nous avons obtenu, sur les possibilités d'évolution de notre routeur et de notre machine virtuelle. Nous envisagerons également les perspectives de recherche que nos travaux ont ouvertes, tant du point de vue de la communication, que sur la gestion des groupes de processus et sur le placement.

1. PAROS/ParX Diffusion Virtual Machine, soit la machine virtuelle à diffusion de PAROS/ParX.

Chapitre 2 : Diffusion et Parallélisme

Notre intention ici n'est pas de couvrir exhaustivement l'ensemble des outils qui font intervenir la diffusion, mais de présenter ceux qui nous paraissent les plus représentatifs (ou populaires!). Le but de cette présentation est de montrer l'importance qu'a pris la diffusion dans le parallélisme.

Nous commencerons par examiner deux langages parallèles où la diffusion est le mode de communication de base : ***Linda*** et ***Esterel*** - le lecteur trouvera dans l'annexe A une présentation d'autres langages parallèles tels que *Ada*, *Occam* ou *FORTRAN*. Puis nous présenterons les systèmes basés sur la diffusion : ***Amoeba***, ***Isis***, ***Vartalaap*** et ***CCL*** - une présentation d'autres systèmes distribués et parallèles est donnée dans l'annexe B. Enfin nous détaillerons les environnements de développement et d'exécution qui offrent des primitives pour les communications globales: ***PVM***, ***p4***, et ***MPI***.

2.1 Langages à diffusion

Notre intérêt vis-à-vis des langages que nous présentons ici est qu'ils adoptent, implicitement ou explicitement, la diffusion comme primitive de communication de base. Dans le cas de Linda nous retiendrons que le synchronisme est la seule véritable contrainte imposée à la diffusion. Par contre l'influence d'Esterel sur nos travaux se porte principalement sur la sémantique de diffusion requise, qui est comparable à celle du modèle CBS.

2.1.1 Linda

Le modèle de programmation parallèle offert par Linda [CaGel89] est construit à partir d'une mémoire associative commune: l'*espace des tuples*. Un *tuple* est objet composé de plusieurs champs de données et peut être de plus associé à un processus, par exemple `("ex",1,1.2)`. Pour manipuler l'espace des tuples seulement six opérations sont définies pour y lire, y retirer ou y écrire des tuples: **`out`**, **`eval`**, **`in`**, **`inp`**, **`rd`** et **`rdp`**.

L'opération `out(tuple)` ajoute un élément à l'espace des tuples; c'est une opération synchrone et atomique. Similairement, `eval(tuple)` crée en plus un processus associé au tuple: le tuple est alors actif jusqu'à la fin de son exécution où il redevient un tuple ordinaire.

Inversement à `out`, l'opération `in(tuple)` recherche un tuple puis le retire de l'espace des tuples. Pour cette opération le tuple donné en paramètre spécifie les éléments de recherche: seuls les champs qui ont une valeur seront pris en compte pour la recherche. Cette recherche prend alors la forme d'un «pattern matching» pour retrouver tous les tuples qui correspondent. Lorsque plusieurs tuples conviennent au tuple spécifié, l'un d'entre eux est choisi de manière non déterministe. S'il n'existe aucun tuple dans l'espace des tuples qui répond aux critères de sélection, l'appel est alors bloquant jusqu'à ce que l'espace des tuples contienne un tuple qui correspond. Une fois sélectionné le tuple est supprimé de l'espace des tuples, et les champs non spécifiés lors de l'appel à `in` prennent alors la valeurs de ceux du tuple.

L'opération `rd` est similaire à `in`, mais le tuple n'est pas supprimé de l'espace des tuples. Les opérations `inp` et `rdp` ont le même rôle que `in` et `rd` respectivement, mais ne sont pas bloquantes.

Pour implémenter Linda, dont le partage de données est le modèle de programmation, sur une machine sans mémoire commune, une première solution, assez naturelle, est de distribuer l'espace des tuples sur l'ensemble des noeuds. La diffusion peut alors être utilisée pour effectuer la recherche des tuples lors des opérations `in` et `rd`. Inversement il est aussi possible de dupliquer l'espace des tuples sur l'ensemble des noeuds. Dans ce cas une diffusion synchrone est nécessaire pour réaliser des changements atomiques et synchrones de tous les exemplaires de l'espace des tuples. D'autres implémentations sont bien sûr possibles, mais quoiqu'il en soit nous voyons bien que la diffusion est très importante pour les réalisations de Linda sur les architectures à mémoire distribuée.

2.1.2 Esterel

Esterel (voir par exemple [BoSi91] ou [Mign94]) n'est pas à proprement parlé un langage parallèle, mais un langage *réactif synchrone*. Cependant une caractéristique très importante des systèmes réactifs est qu'ils sont intrinsèquement parallèles, et Esterel est en cela un langage d'expression du parallélisme. Ses possibilités dans ce domaine en font d'ailleurs un outil pour la spécification, le développement et la validation de protocoles de communication [CCD94].

Un système réactif, et par voie de conséquence un programme Esterel, peut être représenté comme une «boîte noire» avec un ensemble de signaux d'entrée et de sortie. Ces systèmes produisent des signaux en fonction des événements qui leur sont soumis, sous forme de signaux eux aussi, par leur environnement. L'objectif est de répondre instantanément et sans délai à chaque «stimulus»; en d'autres termes les signaux d'entrée et de sortie sont émis simultanément, bien que causalement dépendants. Il s'en suit un *synchronisme parfait* entre un événement et la réaction du système. L'ensemble des calculs qui conduit à une réaction est donc atomique, et le temps n'est alors rythmé que par les stimuli de l'environnement.

Pour traiter en temps réel tous les signaux qui lui sont envoyés, un programme Esterel est constitué d'un ensemble de sous-systèmes concurrents. Du fait de la diffusion et du synchronisme, tous reçoivent tous les signaux en respectant l'ordre temporel ainsi que les dépendances causales. Les signaux qu'ils peuvent générer leurs permettent également de communiquer.

Un signal est associé à un état: présent ou absent, et à une valeur; mais comme le langage ne définit pas de réelles structures de données cette valeur est opaque pour le programme - les signaux à valeur nulle sont dits «purs». L'utilisation de structures de données se fait en général dans un autre langage, comme C ou Ada pour lesquels un traducteur de code Esterel existe.

Esterel ne dispose que d'un seul opérateur d'expression du parallélisme: «||». Les deux flots d'instructions s'exécutent en parallèle et la construction ne se termine que lorsque ces deux flots ont pris fin. La sémantique de cet opérateur est donc similaire à celui de CSP, tout comme celle du constructeur PAR d'Occam.

Six opérations élémentaires sont définies pour la manipulation des signaux:

- `emit` *S*: diffusion d'un signal;
- `?` *S*: lecture de la valeur d'un signal;
- `present` *S* `then` P_1 `else` P_2 `end`: test de la présence du signal *S*, et

exécution conditionnelle de P_1 ou P_2 selon que le signal S est présent ou absent;

- `do` P `watching` S: exécution *gardée* par le signal S; cette construction prend fin soit lorsque P se termine, soit une fois l'exécution de P entamée, lorsque S est présent ce qui interrompt P;
- `signal` S `in` P `end`: déclaration du signal S dont la portée se limite à P. Les émissions par l'environnement d'un signal de même nom n'ont aucun effet sur ce nouveau signal, et inversement les émissions de S par P n'ont pas d'incidence à l'extérieur de P.

Pour des raisons de commodité et de simplicité dans l'écriture de programmes, le langage a été étendu avec de nouvelles instructions, qui toutes peuvent se réécrire à l'aide des opérateurs standards. Les principales instructions dérivées sont les suivantes:

- `await` S: qui exprime l'attente bloquante du signal S;
- `do` P `watching immediate` S: pour prendre en compte S avant même que P ne soit commencé;
- `do` P_1 `watching` S `timeout` P_2 `end`: lorsque la garde arrive à échéance, c'est-à-dire quand P_1 prend fin sans que S n'ait été présent, la construction se termine par l'exécution de P_2;
- `do` P `upto` S: P s'exécute tant que S est absent, et si P se termine alors la fin du constructeur est différée jusqu'à l'émission de S;
- `loop` P `each` S: cette boucle exécute indéfiniment P, mais P est redémarré à chaque occurrence de S;
- `every` S `do` P `end`: de fonctionnement identique au constructeur précédent, celui-ci conditionne la première exécution de P par l'émission de S.

Enfin, l'attente simultanée sur plusieurs signaux s'exprime de la manière suivante:

```
await

    case S1 do
        P1

    case S2 do
        P2

    ...

    case Sn do
        Pn

end await
```

2.2 Systèmes tolérants aux défaillances basés sur la diffusion

Bien que cette thèse ne traite pas des problèmes de tolérance aux pannes, qui feraient l'objet d'une autre thèse, l'étude de ces systèmes va nous permettre de dégager à la fois les protocoles requis et les structures de groupes nécessaires. Notre objectif ici est de prendre en compte les contraintes de tels systèmes dans la réalisation de notre machine virtuelle à diffusion, afin qu'elle soit apte à être adaptée à ces systèmes.

2.2.1 Amoeba

Amoeba [MRT90, TMR86, Mull88] est un système distribué dont le modèle de processus est composé de ***processus*** et de ***threads*** et où la diffusion est un élément fondamental. Les threads sont les unités d'exécution séquentielles et les processus encapsulent un ensemble de threads dans un même espace d'adressage. L'échange de messages est ici une fonction primordiale du micro-noyau, et le ***port*** l'unique objet de communication. Minimal, le micro-noyau d'Amoeba se limite à gérer processus, communications, mémoires, et entrées/sorties de base sur les périphériques bruts. Tout autre service est construit au-dessus comme un serveur. Enfin Amoeba est fortement orienté vers la programmation par objets, et dispose pour cela de fonctions de gestion d'objets.

La première particularité d'Amoeba concerne la sémantique associée aux ports. Ils sont dans ce système les vecteurs de transport des messages, mais pour l'utilisateur les communications se font en mode *Client-Serveur* à partir de primitives d'appel de procédures à distance (RPC), primitives qui cachent la présence des ports aux programmeurs. Le figure 2.1 illustre le déroulement d'un appel: invocation de la procédure avec identification de l'opération (`op`), de ses paramètres (`params`) et du service (`cap`); puis attente de la réponse du serveur qui provoquera la reprise de l'exécution. L'un des threads du serveur reçoit la requête du client (`get_request`), effectue l'opération désirée puis retourne le résultat au client (`send_reply`).

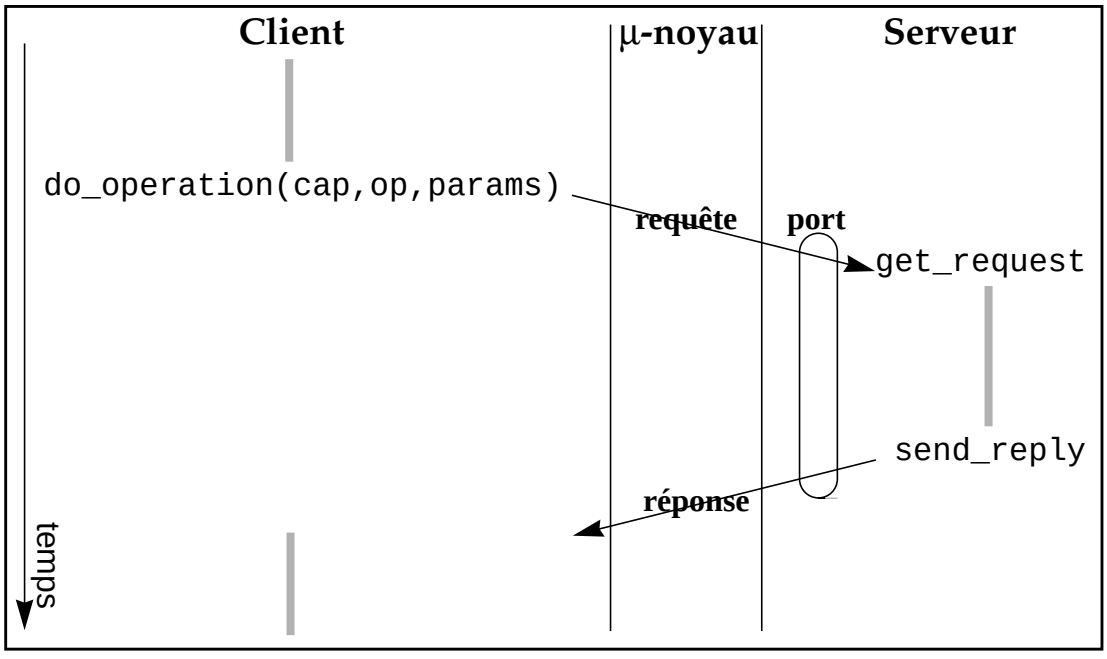


Figure 2.1 : appel de procédure à distance dans Amoeba.

Associé à un processus le port, identifié par une capacité (`cap`) que chaque appel doit spécifier, permet l'accès aux divers services d'un serveur. La capacité est utilisée par le noyau pour localiser le port et par voie de conséquence le serveur, mais elle sert également au service d'authentification pour assurer la protection et la sécurité du système.

La manipulation des capacités, leur structure et les mécanismes de protection qui leurs sont associés différencient également Amoeba des autres systèmes répartis. Si pour des serveurs publics les ports sont accessibles par tous, un port est en général secret et un processus ne peut s'y adresser que lorsque sa capacité lui a été transmise au préalable. A ce premier niveau de protection, un deuxième vient s'ajouter, qui a pour but de se prémunir contre la divulgation des capacités réelles des ports, celles qui servent aux serveurs à lire le contenu de leurs ports (`get_request(cap`$_S$`)`, ...). Pour cela lorsqu'une capacité doit circuler dans le système elle subie une transformation *F* non inversible. De ce fait, la capacité employée par les clients: `cap`$_C$ = *F*(`cap`$_S$), ne peut être détournée pour accéder au port autrement que ce qu'il est seulement permis aux clients. Enfin, un champ «check» supplémentaire dans la structure des capacités (voir figure 2.2), crypté, garantit de surcroît une protection contre la reconstruction de capacités par des entités externes.

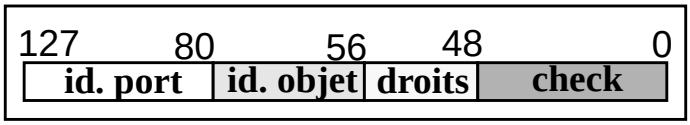


Figure 2.2 : structure d'une capacité dans Amoeba.

Mais la grande particularité d'Amoeba se situe dans sa vision de la répartition de la mémoire dans un réseau. En fait le réseau n'est vu ici que comme un bus pour lequel l'accès aux données est simplement plus lent (de l'ordre du millième). De plus la mémoire partagée n'est pas ici composée d'un ensemble de pages; dans l'orientation objet d'Amoeba cela ne serait pas efficace: la présence de plusieurs objets dans une même page augmenterait le trafic des pages, et ne considérer qu'un objet par page introduirait des copies de mémoires inutiles lorsque la taille d'un objet n'est pas un multiple de celle des pages. Pour ces raisons la mémoire est considérée comme un ensemble d'objets, et c'est à ce niveau que le partage et la cohérence sont assurés.

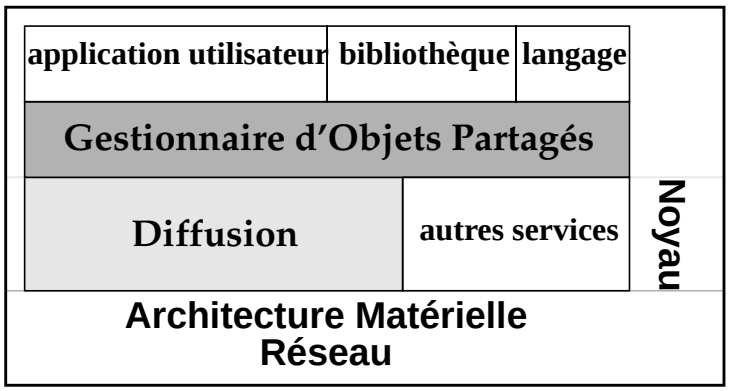


Figure 2.3 : implantation du gestionnaire d'objets partagés d'Amoeba.

Du point de vue de la réalisation chaque site dispose d'un cache relié au réseau comme s'il l'était par un bus, dont la cohérence est garantie par un protocole d'invalidation en écriture, similaire de ceux qui sont mis en oeuvre dans les architectures parallèles à base de bus. Pour que ce protocole d'invalidation et que le réseau soit assimilé à un bus, le micro-noyau

d'Amoeba est doté d'un mécanisme de diffusion fiable et totalement ordonnée [TKB92, KTV92]. C'est d'ailleurs la seule fonctionnalité à faire réellement partie du noyau, le gestionnaire d'objets partagés étant en effet construit au-dessus comme l'illustre la figure 2.3 ci-dessus.

2.2.2 Isis

Isis [BSS91b, BCG91] est un système distribué destiné à la manipulation de groupes de processus, et à la communication au sein d'un groupe. Son objectif principal est d'offrir un support d'exécution fiable qui préserve un état cohérent des groupes. Isis perçoit un groupe comme un «petit» ensemble distribué de processus, et une application comme la réunion d'un «grand» nombre de ces groupes.

La gestion des groupes est adaptée à différents types d'utilisations:

- pour ***offrir un service à des clients***. Dans ce cas les membres du groupe sont des serveurs qui fournissent des services à des processus clients, soit dans un mode de type requête-réponse, soit sous forme de messages de rappels qui sont générés pour indiquer des changements d'états et envoyés aux clients enregistrés;
- pour la ***distribution ou la réplication d'objets***. Dans ce type de groupes, les processus coopèrent pour gérer la concurrence des accès à un objet[1], sa cohérence, ou la tolérance aux pannes;
- pour des ***services systèmes distribués et tolérants aux fautes***. Ce type de services, très rarement explicitement employés par les applications, s'intègre dans le support d'exécution pour des tâches de contrôle, d'observation ou de déverminage.
- pour une ***tolérance aux fautes transparente***. Ici, tous les composants d'un système distribué sont systématiquement remplacés par un groupe de processus tolérant aux fautes.

L'architecture d'implantation d'Isis rend cette gestion indépendante à la fois du matériel et du système d'exploitation. Comme nous pouvons le voir sur la figure 2.4, cette architecture comprend une bibliothèque de fonctions attachée à l'application, qui donne accès à un noyau de communication et de gestion des groupes construit au-dessus du système d'exploitation.

Un groupe est un simple ensemble d'*extrémités de communication*: processus émetteurs et récepteurs, qui peuvent ainsi être référencées comme une seule entité. De telles extrémités peuvent aussi bien être des adresses de sockets sous Unix, des ports sous Chorus, ou encore des capacités dans Amoeba. Cette implémentation des groupes permet à Isis de rester accessible dans un environnement hétérogène de systèmes. Ainsi constitués, les groupes peuvent avoir plusieurs structures: ***fermée***, ***client-serveur***, à ***diffusion*** et ***hiérarchique***, qui correspondent à leur utilisation (cf. figure 2.5). Il existe d'autres structures de groupes, qui sont des formes dégénérées des précédentes et font appel à des implantations particulières; nous ne les détaillerons pas ici.

1. un objet est un composant de l'application.

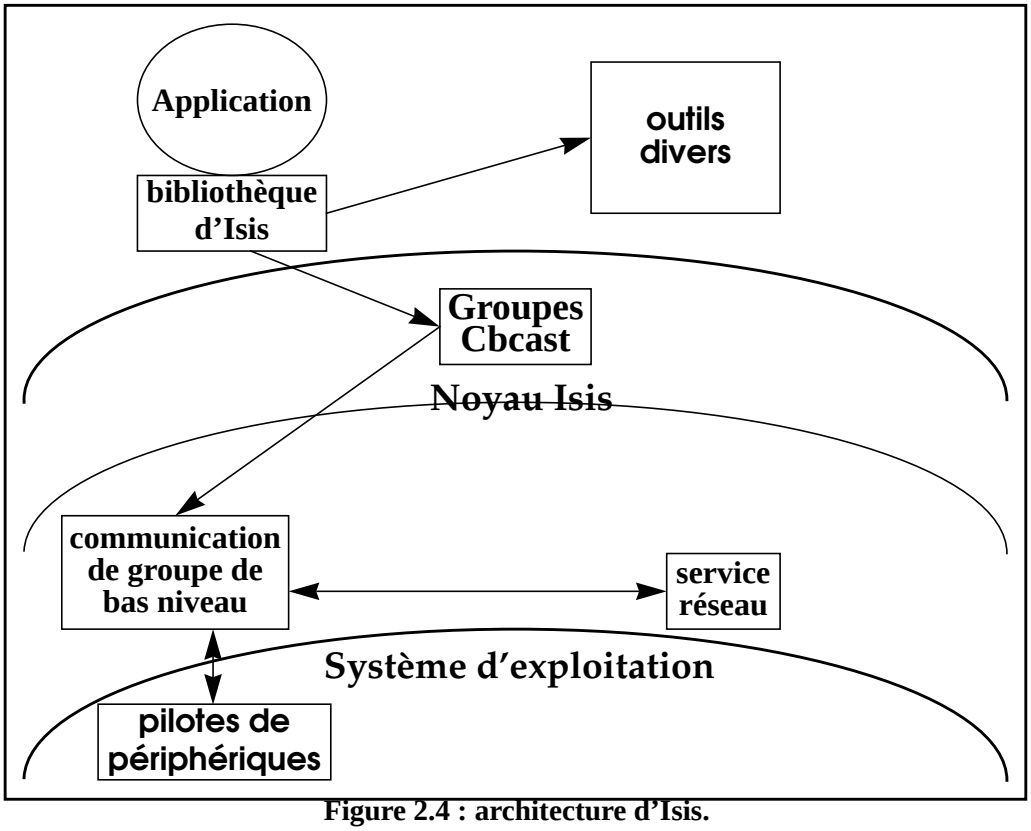


Figure 2.4 : architecture d'Isis.

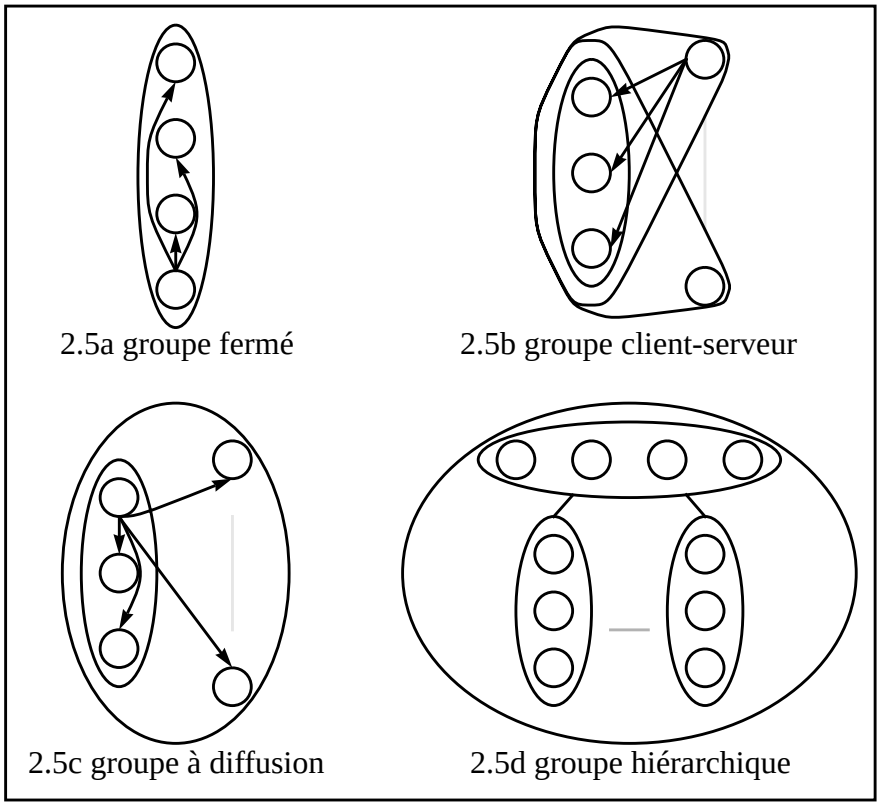


Figure 2.5 : principales structures de groupes dans Isis.

Au sein d'un groupe fermé (figure 2.5a), généralement de petite taille, les membres coopèrent étroitement pour accomplir les mêmes tâches. Le but de tels groupes est d'offrir des services tolérants aux pannes, avec une répartition de charge.

Un groupe de type client-serveur (figure 2.5b) est un groupe fermé de serveurs, auxquels s'adressent des clients soit par diffusion, soit par appel de procédures à distance (RPC) auprès de serveurs donnés. Les serveurs peuvent répondre à un client soit par un échange de messages point-à-point, soit par diffusion pour qu'un ou plusieurs autres processus puissent recevoir atomiquement une copie de la réponse. Cette dernière possibilité trouve son intérêt dans la tolérance aux pannes, où un serveur de sauvegarde sera de cette manière tenu au courant du succès des requêtes des clients - ce succès est marqué par la réception d'une copie de la réponse qui, du fait de l'atomicité dans la communication, marque la réception de la réponse par le client - et des requêtes non encore servies.

Les groupes à diffusion (figure 2.5c) sont des cas particuliers de groupes client-serveur. A l'intérieur de ceux-ci un message peut être envoyé par un serveur vers tous les autres serveurs et vers la totalité des clients. Cette situation est, pour Isis, celle qui fait intervenir le plus de destinations pour une même communication.

Dans des applications importantes, la conception de groupes de grande taille se fait par une décomposition en sous-groupes; c'est cela que modélisent les groupes hiérarchiques (figure 2.5d). Suivant l'optique client-serveur, un client est associé, de manière transparente, à un sous-groupe qui acceptera ses requêtes; cette association, qui est dynamique, est réalisée par le groupe racine. Pour le client le comportement d'une telle structure est le même que dans le cas du client-serveur. La diffusion à l'ensemble des sous-groupes est possible, quoique cela soit rarement nécessaire.

Dans l'approche client-serveur suivie par Isis, un processus serveur peut être le client d'un groupe de serveurs, et, un client peut avoir plusieurs serveurs. De ce fait un processus peut appartenir à plusieurs groupes à la fois, et la composition d'un groupe peut recouvrir celle d'un autre. La composition d'un groupe est de plus dynamique dans Isis, ce qui permet à un processus de se joindre ou de quitter un groupe à tout moment lors de l'exécution.

Pour assurer la cohérence des données, et notamment celle de la composition d'un groupe, ainsi que pour garantir la tolérance aux pannes, les opérations définies par Isis sont **atomiques**. C'est-à-dire que rejoindre un groupe, en sortir, diffuser un message ou faire appel à un service sont des opérations atomiques. Pour réaliser cette atomicité, le support d'exécution possède deux protocoles de diffusion: ABCAST et CBCAST [JoBi88], qui constituent le coeur d'Isis.

Le premier, inclu dans le noyau d'Isis, délivre les messages selon un ordre total, où tous les récepteurs reçoivent les messages dans le même ordre. Le second implémente un ordre causal pour lequel l'ordre de livraison est l'ordre d'émission en prenant en compte les relations de causalité et de précédence entre les messages.

L'interface de programmation de groupes d'Isis est assez réduite et comprend seulement neuf primitives de haut niveau, et six de bas niveau. Comme nous ne nous intéressons dans ce chapitre qu'aux fonctions de manipulation des groupes et de communication par diffusion, nous ne retiendrons que les sept primitives essentielles d'Isis. Comme nous pouvons le remarquer dans le tableau 2.1, les deux protocoles de diffusion sont présents dans les fonctions de haut niveau accessibles aux utilisateurs, ainsi que d'autres comme FBCAST[1] [BSS91a] que nous n'avons pas inclus dans le tableau 2.1. Il n'y a pas de primitives de réception autres que celles du système sous-jacent. Le reste de la bibliothèque est consacré à la manipulation des groupes.

1. Fast Causal Multicast.

primitive	fonction
`gid = pg_create(membres)`	création d'un groupe avec des membres initiaux
`pg_add(gid,pid,type)`	ajout d'un membre (client ou serveur)
`pg_del(gid,pid,type)`	suppression d'un membre (client ou serveur)
`pg_monitor(gid,proc)`	définition de la procédure appelée à chaque modification du groupe
`membres = pg_get_view(gid)`	composition d'un groupe
`pg_entry(gid,proc)`	définition de la procédure appelée à chaque livraison d'un message
`cbid = abcast(gid,msg)`	diffusion ordonnée d'un message à un groupe
`cbid = cbcast(gid,msg)`	diffusion causale d'un message à un groupe

Tableau 2.1 : interface de haut niveau d'Isis.

2.3 Systèmes à diffusion pour le gain en performance

Les deux systèmes que nous présentons représentent deux tendances, l'une orientée réseaux de machines, et l'autre orientée parallélisme. C'est plus particulièrement à ces types de systèmes que s'adresse notre machine virtuelle en tant que support d'implémentation. Leur étude va donc nous permettre de définir une structure de réalisation commune.

2.3.1 Vartalaap

Vartalaap [LRMM93] se présente comme un support de système distribué conçu pour les communications par diffusion, dont l'implémentation est indépendante du matériel. Comme pour Isis, la diffusion est sélective et se fait au sein de groupes de processus.

Il comprend un unique protocole de communication dont l'objectif majeur, hormis celui de délivrer un message à tous ces destinataires, est d'optimiser les échanges de messages afin de surcharger le moins possible le réseau. En effet, plus spécifiquement développé pour les réseaux locaux, Vartalaap considère le médium de communication comme un goulot d'étranglement. L'optimisation touche également les sites connectés au réseau qui ne doivent pas recevoir de messages non destinés à un processus local.

La figure 2.6 schématise, sur un exemple, la minimisation des échanges de messages. Dans cet exemple quatre sites *A*, *B*, *C* et *D* sont connectés au réseau, un processus p_1 du site *A* envoie un message aux processus p_2 et p_3 placés sur le site *C*, ainsi qu'au processus p_4 en *D*. Pour réaliser cette communication il est inutile, comme cela est représenté dans la figure 2.6a, que le site *B* réceptionne le message, et que le message soit envoyé deux fois à *C* pour atteindre p_2 et p_3. Le schéma de communication, dans cette situation, mis en oeuvre par Vartalaap est présenté dans la figure 2.6b et consiste à n'envoyer que deux messages à travers le réseau : le premier vers le site *C* qui transmettra une copie à p_2 et p_3, et le second vers *D* pour atteindre p_4.

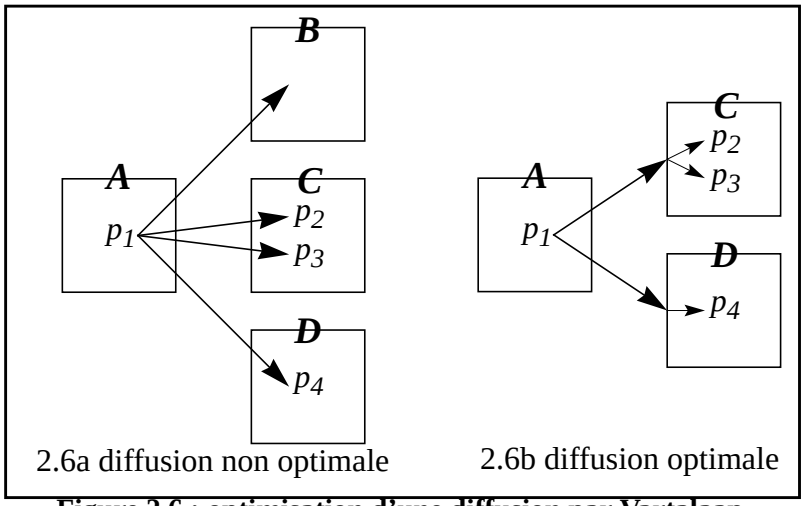


Figure 2.6 : optimisation d'une diffusion par Vartalaap.

Pour obtenir un tel comportement de communication, l'architecture conceptuelle de Vartalaap est composée de trois éléments: une bibliothèque de fonctions, un serveur local placé sur chaque noeud, et un serveur global situé dans l'une des machines du réseau. La figure 2.7 présente les différents liens qui relient ces éléments.

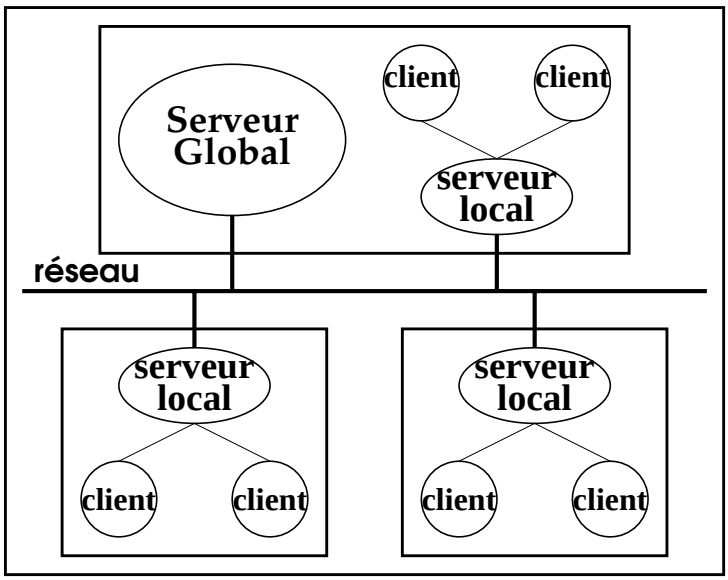


Figure 2.7 : architecture générale de Vartalaap.

La bibliothèque lie un client au serveur local, elle est chargée du découpage des requêtes en paquets et de leur réassemblage. Ensuite vient le serveur local qui distribue les messages aux processus locaux par l'intermédiaire d'une table locale qui associe les processus à leurs groupes, localise le serveur global et lui transmet les messages. Enfin, le serveur global est chargé de distribuer les messages aux serveurs locaux et de gérer l'hétérogénéité des matériels. Pour assurer sa fonction principale d'optimisation des communications, le serveur global conserve une table de répartition des groupes, table dans laquelle pour chaque groupe est indiqué, en plus d'un nom logique ou *alias*, l'ensemble des sites occupés par ses membres. Cet enchaînement des actions dans une communication est schématisée par la figure 2.8.

Outre les échanges de messages point-à-point réalisés avec les outils de communication du système sous-jacent (par exemple les sockets TCP/IP d'Unix), les serveurs implantent un protocole de diffusion. Il s'agit d'un protocole asynchrone simple, qui envoie un message pour chaque noeud destinataire.

L'interface de ce système à diffusion comporte dix primitives que nous pouvons diviser en trois classes: initialisation et terminaison, manipulation des groupes, et communication. Le tableau 2.2 présente l'ensemble de ces fonctions ordonnées selon leur classe.

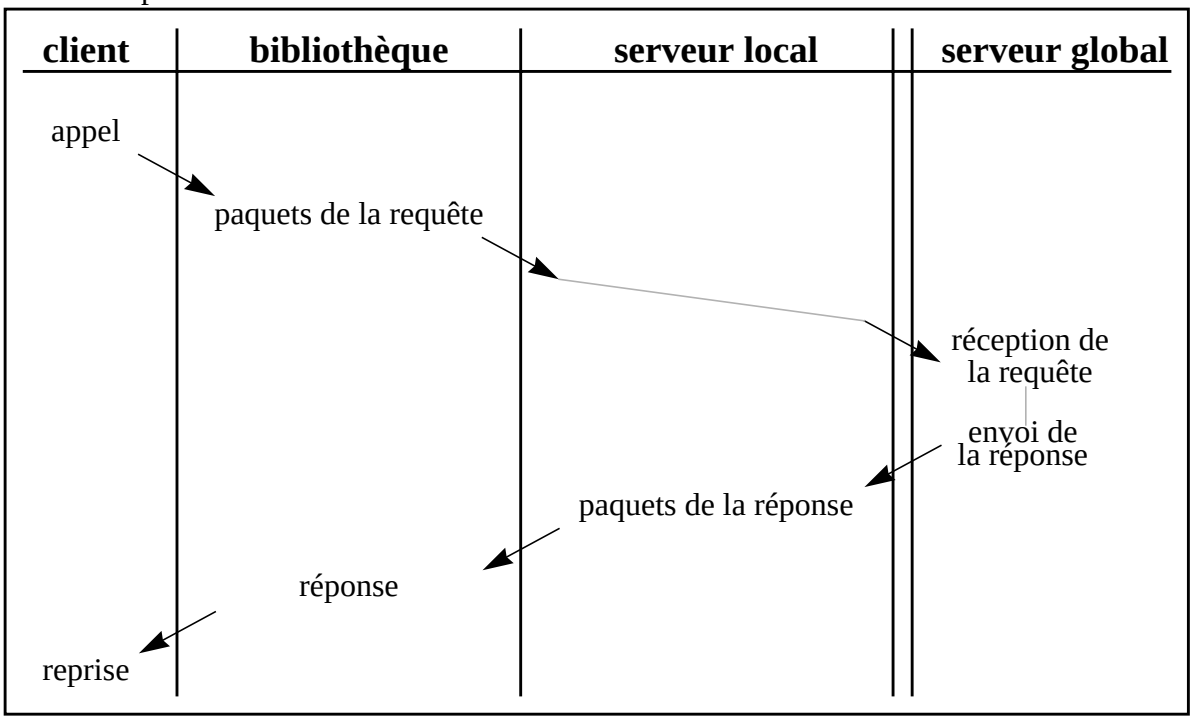


Figure 2.8 : cheminement d'un appel dans Vartalaap.

La manipulation des groupes est assez générale, au sens ou un processus peut appartenir à plusieurs groupes en même temps, et qu'il peut rejoindre ou quitter un groupe à tout moment de son exécution. Par contre l'emploi des primitives de communication suit des règles spécifiques à ce système. D'une part l'envoi d'un message à un groupe ne peut se faire que si l'expéditeur est membre du groupe; l'appel à la primitive est par ailleurs asynchrone et retourne dès que le message a été expédié (dans le cas ou le groupe existe). D'autre part, la réception ne concerne pas forcément qu'un seul message à la fois, mais peut être relative à plusieurs messages; elle est de surcroît soumise à une fonction de filtrage des messages donnée par l'utilisateur. Contrairement à l'envoi non bloquant de messages, le retour d'un appel à la primitive de réception est conditionné par la valeur du paramètre `time`:

- **`VPoll`** spécifie de ne pas attendre lorsque les tampons sont vides, si bien que seuls les messages déjà arrivés au site sont concernés;
- **`VForever`** bloque le processus jusqu'à ce qu'il y est au moins un message reçu qui corresponde au filtre;
- toute autre valeur entière et positive spécifie un délai d'attente maximum (exprimé en secondes).

primitive	fonction
`VInitTalkLib(...)`	connexion serveur local initialisation bibliothèque
`VShutTalkLib()`	déconnexion
`VMakeNewGroup(gid)`	création nouveau groupe créateur membre
`VJoin(gid)`	rejoindre un groupe
`VLeave(gid)`	quitter un groupe
`VAliasGroup(gid,alias)`	donner un nom à un groupe
`VUnAliasGroup(alias)`	supprimer le nom de groupe
`VGetId(Alias,gid)`	retrouver l'identificateur d'un groupe à partir de son nom
`VSend(gid,msg,size)`	expédier un message à un groupe
`VReceive(gid,msgList,nbMsg,filter,time)`	réception d'un ensemble de messages

Tableau 2.2 : interface de Vartalaap.

2.3.2 CCL

A la différence de ces prédécesseurs, CCL [BBCE94] n'est pas un support de communication mais une bibliothèque de fonctions pour les communications de groupes. Elle n'a pas été non plus conçue pour des réseaux locaux mais pour des architectures parallèles - sa première implantation s'est faite sur les machines *IBM **SP1*** et ***SP2***. Elle repose sur un modèle de communication point-à-point où tous les noeuds sont deux à deux connectés.

La définition du modèle sur lequel s'appuie CCL est suffisamment simple pour que l'implémentation de la bibliothèque reste possible avec la plupart des noyaux de communications. Il comprend une primitive d'envoi de messages (**send**), une autre de réception (**receive**), ainsi qu'une fonction d'émission et de réception cumulées (**send_receive**[1]). Ce sont des opérations asynchrones qui préserve l'intégrité des données: un **send** bloque le processus appelant jusqu'à ce que les données soient intégralement copiées de l'espace mémoire utilisateur; de même le retour d'un **receive** n'intervient qu'une fois le message entièrement copié dans l'espace mémoire utilisateur. L'usage de tampons dans le système de communication sous-jacent n'est pas obligatoire, mais il est impératif qu'il respecte en réception l'ordre d'émission (vis-à-vis d'une même source). De plus les émissions de messages par des sources différentes ne doivent pas interférer les unes par rapport aux autres, afin que, quels que soient les messages en attente, un processus puisse toujours recevoir un message d'une source lorsqu'il existe. Enfin,

1. sa réalisation doit être correcte, i. e. elle ne fait pas intervenir un espace tampon dynamique et prévient les interblocages.

la protection des communications doit être garantie de telle sorte que les échanges de messages produits par CCL restent internes à la bibliothèque.

La notion de groupe est ici plus proche de celle de MPI: un groupe est ensemble ordonné de processus qui possède un nom unique; i. e. tout membre possède un rang dans ce groupe. Nous ne retrouvons cependant pas tous les opérateurs ensemblistes de MPI, seule la partition est disponible. Hormis cette possibilité, la création d'un groupe se fait à partir de la liste explicite de ses membres. Un groupe n'est donc pas ici une entité dynamique dont les participants peuvent évoluer dans le temps. Par contre un processus peut appartenir simultanément à plusieurs groupes ou partitions.

CCL définit un groupe par défaut, **`ALL`**, qui comprend tous les processus de l'application. Tout autre groupe est créé par un appel à la primitive **`group`**:

```
gid = group(size,list,label)
```

Tous les participants du nouveau groupe doivent invoquer cette fonction, avec les mêmes paramètres. L'ordre dans le groupe est celui donné dans la liste de ses membres (`list`). La partition d'un groupe `G` en sous-groupes est effectuée de manière similaire: tous les membres de `G` invoque l'instruction suivante:

```
gid = partition(G,val,key)
```

Lors de cet appel il y a synchronisation et les partitions déterminées par les différentes valeurs de `val` données par les processus. Le paramètre `key` permet de définir le rang de chaque processus dans le sous-groupe auquel il va appartenir.

Au sein d'un groupe les processus participent à des actions globales qui engagent le groupe entier. Ces actions, qui se traduisent par des échanges de messages, ne prennent fin que lorsque chacun a joué son rôle. Certaines d'entre elles imposent une barrière de synchronisation, soit une synchronisation forte, alors que d'autres sont asynchrones et permettent aux processus de reprendre leur exécution une fois leur part de travail effectuée.

primitive	type	fonction	synchronisation
`bcast`	1 vers tous	diffusion de messages	non
`reduce`	tous vers 1	réduction	non
`combine`	tous vers tous	réduction	forte
`scatter`	1 vers tous	dispertion	non
`gather`	tous vers 1	rassemblement	non
`concat`	tous vers tous	diffusion	forte
`index`	tous vers tous	personnalisée	forte
`prefix`		balayage avec réduction	non
`shift`	1 vers 1	permutation cyclique	non
`sync`		barrière de synchronisation	forte

Tableau 2.3 : opérations globales de CCL.

La bibliothèque des opérations de groupe est similaire à celle de MPI. L'ensemble de ses primitive est résumé dans le tableau 2.3. Quel que soit le rôle d'un processus dans l'une de ces opérations, il doit invoquer la même fonction, avec les mêmes paramètres. Par exemple, l'émetteur et les récepteurs d'une diffusion exécutent l'instruction:

```
bcast(G,orig,data)
```

Le diffuseur est identifié dans le paramètre `orig`, et seul ce processus émet les données `data`; les autres participants reçoivent ces données dans `data`. De même, pour effectuer une combinaison tous les membres d'un groupe fournissent leurs données (`data`), la fonction de combinaison (`rfunc`); puis ils appellent la primitive `combine`:

```
combine(G,rfunc,data,result)
```

et reçoivent le résultat dans `result`.

2.4 Environnements de développement et d'exécution

Ces dernières années une autre approche du parallélisme a émergé et connu un franc succès auprès des utilisateurs: les environnements de développement et d'exécution d'applications parallèles. Le fait qu'ils soient simples à mettre en oeuvre et à utiliser, qu'ils soient disponibles pour la plupart des machines séquentielles et pour des réseaux de machines, explique leur popularité et leur grande distribution.

Le but recherché est de faciliter le plus possible la programmation parallèle, que l'utilisateur soit chevronné ou qu'il soit néophyte. Pour cela divers outils sont fournis, pour la gestion et le contrôle de l'environnement dont dispose l'utilisateur, pour le développement et l'aide au déverminage d'applications. Tous ces outils sont axés sur une bibliothèque de primitives - généralement disponible pour plusieurs langages, et principalement pour C et FORTRAN -, qui seule suffit à la programmation.

La vision du parallélisme qu'offre de tels environnements est celle d'un réseau de machines séquentielles et/ou parallèles accessibles et exploitables avec les outils classiques. Ainsi ils sont capables de fonctionner avec des réseaux hétérogènes et les applications produites sont de ce fait portables.

Parmi tous les environnements généraux existants nous avons choisi **PVM** qui est le plus connu et sans doute le plus employé, **p4** qui diffère légèrement du premier et propose des fonctions plus variées, et **MPI** le plus récent et qui tend à remplacer PVM. Tout comme les systèmes précédents, c'est avec la majorité de ces environnements que notre machine virtuelle doit rester compatible.

2.4.1 PVM

La philosophie de PVM [GBD93] est, comme son nom l'indique[1], de donner à l'utilisateur la vision d'une *machine parallèle virtuelle* à mémoire distribuée composée d'un ensemble de machines séquentielles, parallèles ou vectorielles. Multi-utilisateur, PVM permet de faire cohabiter plusieurs machines virtuelles simultanément; les seules restrictions sont celles imposées par l'exploitation des machines réelles - c'est le cas par exemple d'un Intel IPSC/860 qui ne peut être subdivisé qu'en 10 partitions au maximum.

1. PVM pour Parallel Virtual Machine (Machine Virtuelle Parallèle).

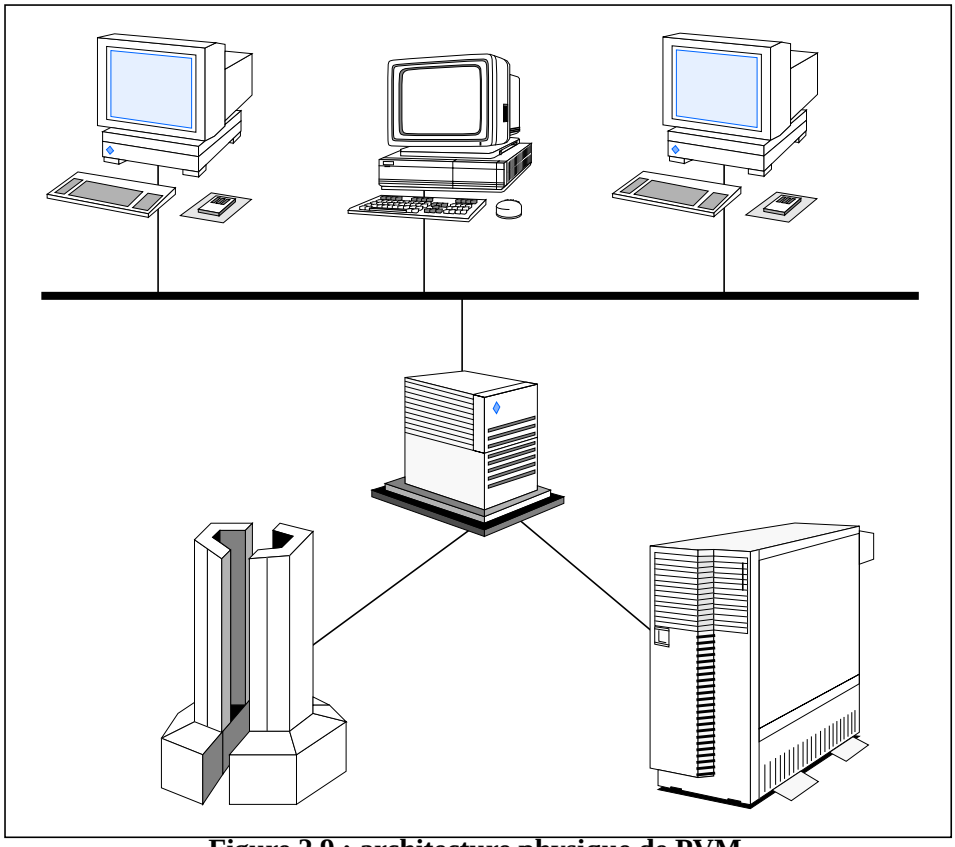


Figure 2.9 : architecture physique de PVM.

Bien que pour certaines interfaces cela ne soit pas obligatoire, l'implantation traditionnelle de PVM se fait sur un réseau de machines sous Unix (cf. figure 2.9); l'exploitation de la plupart des calculateurs parallèles, dépourvues d'une version de ce système, est assurée à partir d'un serveur Unix hôte. La définition d'une machine virtuelle se réfère toujours aux ordinateurs connectés au réseau. Ceci est spécifié dans un fichier de configuration qui dresse simplement la liste de ceux qui appartiennent à la machine virtuelle, ainsi que ceux qui, sans en être exclu, pourront au cours de l'exécution y être adjoints.

A partir d'une machine virtuelle l'utilisateur développe son application comme un simple ensemble de processus et de tâches. Ne possédant pas de hiérarchie de processus, le modèle de processus de PVM se résume à la simple distinction entre les processus Unix classiques et les tâches. Tous les deux sont des unités de calcul séquentielles, mais à la différence des processus Unix, les tâches ont accès aux primitives de PVM, ce qui les rend indépendantes d'Unix et aptes à être exécutées sur une architecture parallèle. Outre l'accès aux possibilités offertes par le système, un processus peut se promouvoir en tâche vis-à-vis de PVM - c'est la seule primitive de la bibliothèque à laquelle il a accès. Pour gérer processus et tâches trois primitives sont fournies: **`pvm_spawn`** qui crée plusieurs tâches identiques, **`pvm_mytid`** qui renvoie l'identificateur de la tâche appelante et fait d'un processus une tâche, puis **`pvm_exit`** qui met fin à une tâche qui redevient alors processus.

Le principal intérêt de cet environnement est d'offrir une interface de communication par échange de messages cohérente avec le paradigme des architectures parallèles. Bien que toutes les sémantiques de communication ne soient pas inclues à la bibliothèque, il offre un ensemble minimal de fonctions. Pour cela le routage et l'acheminement des messages sont pris en charge par le support d'exécution de PVM, et sont donc transparents pour l'utilisateur. Deux types de

communication sont considérés: l'échange de messages point-à-point asynchrone, et la diffusion asynchrone.

Puisque le réseau est supposé hétérogène, un mécanisme de conversion entre les différentes représentations de données encapsule les communications. Ce mécanisme n'est malheureusement pas transparent à l'utilisateur qui, pour envoyer (resp. recevoir) des données, doit empaqueté (resp. désempaqueté) son message. De même la gestion des tampons, tâche délicate de tout système de communication asynchrone, est reporté au niveau de l'utilisateur: c'est à lui d'allouer et libérer l'espace mémoire nécessaire. Toutefois en réception, du fait de l'asynchronisme, l'allocation des tampons est automatique. PVM fixe les règles de gestion qui doivent être respectées:

- l'émission de plusieurs messages se fait séquentiellement, c'est-à-dire qu'il ne peut y avoir qu'un seul tampon actif à la fois;
- dès réception d'un message le tampon précédent est automatiquement libéré, à moins qu'il n'ait été au préalable explicitement sauvegardé.

Ainsi l'émission d'un message se fait en trois étapes:

1- allocation d'un tampon;

2- empaquetage du message dans le tampon;

3- expédition du message.

Inversement la réception d'un message par une tâche se déroule en deux phases:

1- réception du message

2- désempaquetage

Pour l'émission deux primitives: **`pvm_send`** et **`pvm_mcast`**, servent respectivement à l'envoi de messages à une seule ou à un ensemble de tâches. Ces deux opérations sont asynchrones, c'est-à-dire que la tâche appelante ne reste suspendue que jusqu'à l'expédition complète du message par le noeud. Afin de permettre la réalisation d'un contrôle de flux, la gestion de priorités ou encore le typage des informations, une étiquette[1] doit être associée à chaque message. Pour spécifier les tâches réceptrices il faut spécifier les identificateurs uniques que PVM associe à chacune d'elles - la diffusion d'un message par `pvm_mcast` se fait donc en spécifiant tous les récepteurs.

Quel que soit le mode d'émission, point-à-point ou diffusion, les mêmes fonctions sont utilisées pour recevoir les messages: **`pvm_nrecv`** pour une réception non bloquante, et **`pvm_recv`** pour une réception bloquante. Parmi les messages disponibles la sélection se fait selon l'identificateur de la tâche émettrice, et selon l'étiquette attendue. Toutes les combinaisons sont possibles suivant que l'un et/ou l'autre des deux critères sont spécifiés ou laissés libres.

Afin de permettre une meilleure coopération entre tâches, notamment pour la synchronisation ou la diffusion, PVM intègre la notion de *groupe* de tâches. Les primitives de manipulation des groupes sont assez générales et n'imposent pas de réelles restrictions. De ce fait la sémantique d'un groupe n'est soumis qu'aux trois règles suivantes:

1. tag dans la terminologie anglaise de PVM.

- **nommage**: chaque groupe est nommé par l'utilisateur;
- **dynamicité**: une tâche peut à tout moment se joindre (**`pvm_joingroup`**) ou quitter (**`pvm_lvgroup`**) un groupe;
- **multiplicité**: une tâche peut appartenir à plusieurs groupes en même temps, sans limitation.

Hormis des fonctions de contrôle de la composition des groupes, deux opérations sont définies sur les groupes: la diffusion de messages à l'ensemble des membres du groupe[1] (**`pvm_bcast`**), et une «barrière» de synchronisation (**`pvm_barrier`**) qui définit un point de rendez-vous entre un certain nombre de participants, quelconques, dans le groupe. La barrière de synchronisation n'est accessible qu'aux membres du groupe, alors que la diffusion peut se faire sans y appartenir.

Une des particularités de PVM est de posséder un autre mécanisme de communication et synchronisation que l'échange de messages. En effet, plutôt que des messages, les tâches peuvent s'envoyer des signaux ou des événements. L'émission de signaux (**`pvm_sendsig`**) est similaire à celui d'Unix, mais ici cela est possible entre des tâches distantes. Par contre la soumission d'un événement (**`pvm_notify`**) se traduit par la diffusion d'un message, si bien que les tâches concernées par l'événement en seront informées lors d'un appel à l'une des primitives de réception.

Enfin, comme complément de sa bibliothèque, PVM offre des outils d'aide au développement, à la détection et à la correction d'erreurs. Pour cela deux modes d'exécution sont possibles: le mode trace et le mode «debug». Le mode trace permet de visualiser le comportement d'une application, tandis que le mode «debug» sert au déverminage d'un programme à l'aide des outils standards disponibles sur les machines du réseau. Toutefois les possibilités de PVM dans ce domaine sont encore insuffisantes, surtout lorsque l'on se place dans le contexte du parallélisme massif.

2.4.2 p4

Comme PVM, p4[2] [BuLu92] se présente sous la forme d'une bibliothèque accompagnée d'outils, et comme PVM l'architecture considérée est un réseau de machines hétérogènes, parallèles ou non, avec Unix comme système d'exploitation sous-jacent. En revanche, il ne propose pas de machine virtuelle, uniforme quels que soient les matériels disponibles,mais les primitives de la bibliothèque sont capables d'exploiter aussi bien les particularités des architectures à mémoire distribuée que celles des multi-processeurs à mémoire commune.

Dépourvu de machine virtuelle, l'utilisateur doit configurer chaque application par rapport au réseau qu'il a sa disposition. Pour cela il lui faut spécifier, dans un fichier qui décrit le réseau, le placement initial des processus: chaque ligne du fichier associe à une machine un nombre d'occurrences du même processus dont elle a la charge. Une même machine peut être référencée plusieurs fois afin d'y autoriser le placement de processus différents. A partir de cette description, l'amorçage d'une application est réalisé par un processus particulier: le *maître*, qui dans un mode proche du maître/esclaves, lance l'exécution des processus de l'applications sur les différentes machines.

1. l'implémentation standard ne garantit pas la cohérence des destinataires, i. e. dans certains cas des tâches qui viennent de quitter le groupe juste avant la diffusion recevront quand même le message, alors que d'autres qui ont rejoint le groupe ne le recevront pas.
2. Portable Programs for Parallel Processors.

Processus Unix, le maître acquitte sa charge de démarrage par un appel à la fonction **p4_create_procgroup**, et il doit être le seul à le faire. En dehors de ce processus et des processus propres à p4, l'utilisateur dispose de deux catégories de processus: ceux dont p4 a la charge et qui peuvent faire appel à ses services, et ceux gérés entièrement par l'application et qui n'ont pas accès aux fonctions de la bibliothèque. Seuls les processus spécifiés dans le fichier de configuration sont pris en compte par p4, tout autre processus créé dynamiquement (**p4_create**) n'est pas contrôlé par p4 mais par l'utilisateur. La coopération et la communication entre un processus utilisateur et son «créateur» ne peut se faire que par l'intermédiaire de zones de mémoires partagées.

Par contre les processus pris en charge par p4 disposent d'un ensemble assez vaste de primitives de communication: des fonctions d'échange de messages point-à-point, des opérations globales, et la mémoire partagée lorsqu'elle est disponible. Pour l'échange de messages la bibliothèque offre divers moyens de contrôle de la gestion des tampons ou de la conversion des données, et deux modes de fonctionnement: asynchrone ou synchrone. Le choix entre toutes les possibilités est fait par le processus émetteur selon la fonction appelée (voir tableau 2.4); la réception est par contre assurée par une seule primitive: **p4_recv**. La réception est une opération toujours bloquante où l'utilisateur ne peut simplement agir que sur la gestion du tampon. Cependant le test de messages en attente est possible (**p4_messages_available**), ce qui permet de redéfinir une réception asynchrone.

Primitive	Mode	Gestion des tampons	Conversion de données
p4_send	asynchrone	automatique	sans
p4_sendr	synchrone	automatique	sans
p4_sendb	asynchrone	utilisateur	sans
p4_sendx	asynchrone	automatique	avec
p4_sendbr	synchrone	utilisateur	sans
p4_sendbx	asynchrone	utilisateur	avec
p4_sendrx	synchrone	automatique	avec
p4_sendbrx	synchrone	utilisateur	avec

Tableau 2.4 : primitives d'envoi de messages de p4.

En complément de l'échange de messages point-à-point, p4 fournit des opérations globales; mais comme il n'y a pas, contrairement à PVM, de notion de groupe, toutes ces opérations agissent sur l'ensemble des processus gérés par p4. A cette fin la bibliothèque est étendue et comprends en plus de la diffusion de messages (**p4_broadcast** et **p4_broadcastx**) et de la barrière de synchronisation (**p4_global_barrier**), une fonction de calcul sur des ensembles de données répartis: **p4_global_op**. Selon le type d'opération (extraction d'un maximum ou d'un minimum, addition ou multiplication) spécifiée, cette primitive applique le calcul sur les données distribuées à tous les processus, puis diffuse le résultat. Enfin une primitive de terminaison correcte est fournit: **p4_wait_for_end**, ce qui évite, par rapport à PVM, de définir un groupe contenant tous les processus et d'y appliquer une barrière de synchronisation en guise de point de terminaison.

La particularité principale de cet environnement est de ne pas considérer le seul modèle à échange de messages, il propose en complément un support minimal pour gérer une mémoire partagée. Plus généralement l'allocation dynamique de mémoire ne doit se faire que par l'intermédiaire des primitives fournit dans la bibliothèque: **`p4_malloc et p4_free`** pour une zone de mémoire non partagée, et **`p4_shmalloc`** et **`p4_shfree`** lorsque la zone de mémoire est partagée.

Pour maintenir la cohérence des informations ainsi partagées, p4 offre les fonctions de base pour réaliser des *moniteurs* selon la forme proposée par **Brinch-Hansen** [Brinch77]. Pour cela cinq primitives sont définies: **`p4_moninit`** pour initialiser un moniteur avec un nombre donné de files d'attente, **`p4_menter`** pour entrer dans un moniteur et **`p4_mexit`** pour en sortir; puis **`p4_mcontinue`** pour libérer, si il existe, le premier processus en attente dans l'une des files d'attente d'un moniteur et **`p4_mdelay`** pour se mettre en attente dans l'une de ces files. Déjà intégrés à cette bibliothèque, quatre moniteurs servent à la gestion de verrous (lock), de compteurs partagés (getsub), de points de synchronisation semblables à ceux de PVM (barrier) et, de répartiteurs de travaux adaptables au problème à traiter (askfor).

2.4.3 MPI

Récemment, face à la fois à la diversité et aux similitudes des environnements de programmation distribuée et parallèle, un forum international, comprenant industriels et universitaires, s'est réuni avec comme objectif la définition d'une interface standard pour l'échange de messages. Aux objectifs communs aux réalisations précédentes, hétérogénéité et indépendance vis-à-vis des langages, viennent s'ajouter des contraintes supplémentaires comme rester proche de ce qui existe déjà, être flexible et proposer des fonctions étendues en rapport avec les nouveaux besoins des développeurs. La collaboration de tous les partenaires a aboutie à une première spécification «stable» de cette interface: MPI[1] [MPI93].

Essentiellement axée sur les communications, MPI définit un ensemble de primitives pour l'échange de messages point-à-point et des opérations globales, suggère un support de réalisation et fixe certaines règles de gestion des messages. Tout ce qui a trait aux outils de développement, à la gestion de la mémoire, aux entrées/sorties n'a pas été considéré. De même aucun modèle de processus n'est proposé; thread[2], tâche ou processus de type Unix, un processus est simplement considéré ici comme une unité d'exécution, dans son sens le plus large. Toutefois, en raison de leur nécessité, groupes et topologies virtuelles de processus sont abordés.

Informations complémentaires aux contextes de communication, les groupes et topologies sont en effet des éléments importants du système de communication: un groupe est simplement un ensemble ordonné de processus, une topologie virtuelle la description d'un graphe de communication, et un contexte l'espace d'échange de messages. Ces trois informations sont encapsulées dans l'objet de communication de MPI: le «communicateur»[3], auxquelles peuvent s'ajouter des attributs propres aux applications. MPI considère en fait la structure d'une application comme un ensemble de groupes de processus pour lesquels il est possible de définir un graphe de communication spécifique. Ces groupes peuvent être disjoints ou non; et dans le cas où ils le sont les communications inter-groupe sont assurées soit par le communicateur d'un groupe englobant, soit par un communicateur spécial, dit inter-communicateur, qui relie deux

1. Message Passing Interface (interface d'échange de messages).
2. du fait de son utilisation plus courante, nous utiliserons le mot anglais thread plutôt que son équivalent en français: flot de contrôle.
3. communicator dans la terminologie de MPI.

processus dans les deux groupes. A l'intérieur d'un groupe les échanges de messages sont simplement gérés par le communicateur du groupe, ou intra-communicateur. La figure 2.10 illustre la vue d'une application par MPI, vue cohérente avec celle que peut en avoir l'utilisateur.

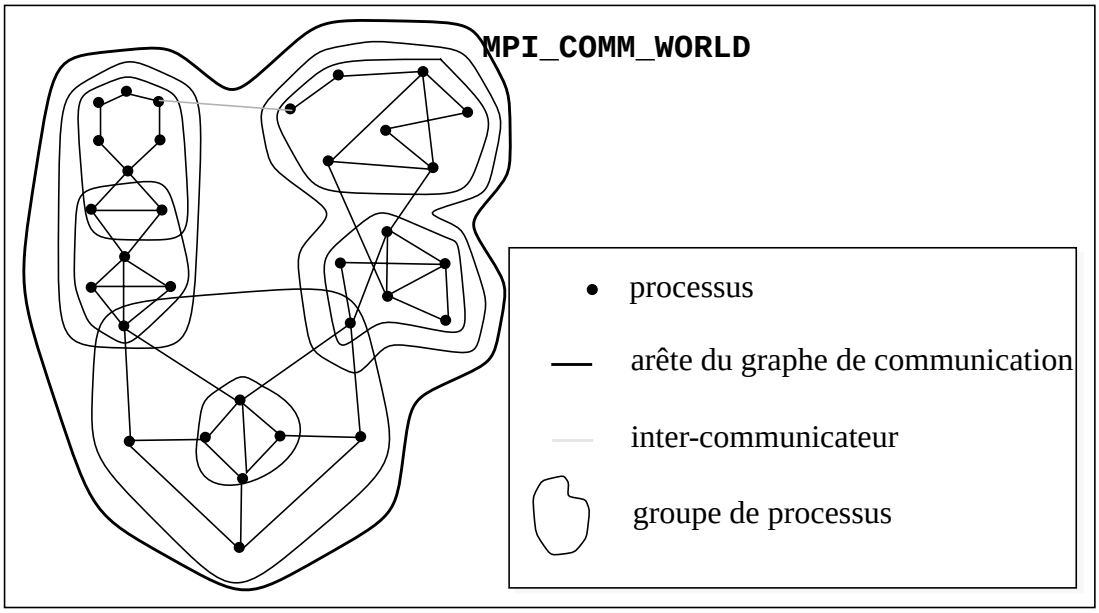


Figure 2.10 : architecture d'une application pour MPI.

Alors qu'à un communicateur est toujours associé un groupe l'inverse n'est pas vrai, si bien que le groupe est une notion plus flexible, et sa gestion est séparée de celles des autres. Puisque les groupes sont considérés comme des ensembles, leur manipulation se fait avec les opérations ensemblistes traditionnelles: l'union (**MPI_GROUP_UNION**), l'intersection (**MPI_GROUP_INTERSECTION**), la différence (**MPI_GROUP_DIFFERENCE**), l'inclusion (**MPI_GROUP_INCL**) et l'exclusion (**MPI_GROUP_EXCL**). Objet du système de communication, les intra-communicateurs se gèrent avec des opérations différentes, à savoir: la duplication totale (**MPI_COMM_DUP**), la duplication sur un sous-groupe (**MPI_COMM_MAKE**), et le partitionnement (**MPI_COMM_SPLIT**). Du fait de sa spécificité, la création d'inter-communicateurs passe par deux phases: la première synchronise tous les processus dans chacun des groupes concernés (**MPI_INTERCOMM_START** ou **MPI_INTERCOMM_NAME_START**), puis il y a synchronisation des deux groupes (**MPI_INTERCOMM_FINISH**). Il est également possible à partir d'un inter-communicateur de créer un nouvel intra-communicateur, et un nouveau groupe associé, composé des deux groupes reliés par l'inter-communicateur (**MPI_INTERCOMM_MERGE**). Toutes ces fonctionnalités font intervenir tous les membres du, ou des groupes concernés; c'est-à-dire que l'opération ne prend effet que lorsque tous y ont contribuée. Enfin un communicateur, **MPI_COMM_WORLD** est prédéfini pour inclure tous les processus de l'application.

Pour le système de communication l'appel à une primitive de communication correspond à la création d'une requête composée du type d'action à effectuer: réception ou émission, et de l'enveloppe qui doit être associée au message. Sorte d'en-tête, l'enveloppe est l'élément de sélection des messages; elle spécifie les paramètres de la communication: source, destination, communicateur et étiquette, qui sont envoyés avec le message. La sélection s'opère en réception, et seuls les champs source et étiquette peuvent être laissés indéterminés. Le système traite ces requêtes, effectue les transferts de données et contrôle l'exécution des processus pour assurer le protocole de communication requis.

Pour des échanges bipartites il existe deux protocoles: *bloquant* et non *bloquant*, et trois modes: *standard*, «*ready*» (prêt) et *synchrone*. Selon le protocole et le mode de communication, une requête peut être:

- *non bloquante*: la procédure se termine avant que la communication ne prenne fin (**MPI_ISEND**, **MPI_ISSEND** et **MPI_IRECV**);
- *localement bloquante*: la procédure ne se termine que lorsque les données mises en jeu peuvent être réutilisées sans conséquence sur le contenu du message (**MPI_SEND**);
- *globalement bloquante*: la procédure ne s'achève qu'après la fin de la communication (**MPI_SSEND**, **MPI_RECV**, **MPI_RECV**, **MPI_SENDRECV** et **MPI_SENDRECV_REPLACE**);
- *soumise à la présence d'une requête correspondante*[1]; ceci n'est valable que pour l'émission vis-à-vis d'une réception dans le mode «ready» (**MPI_RSEND** et **MPI_IRSEND**).

Quel que soit le mode de communication ou le protocole à mettre en oeuvre, le communicateur assure le traitement complet de tous les messages; il est donc polyvalent. C'est à ce niveau que sont fixées les règles sémantiques de gestion des messages:

- MPI garanti une ordre déterministe relatif à la source: à l'intérieur d'un communicateur il ne peut se produire de dépassement de messages;
- MPI assure que l'une ou l'autre des actions d'échange d'un message, émission ou réception, se terminera lorsque ces actions correspondent. L'émission terminera, à moins que la réception soit satisfaite par un autre message. La réception s'achèvera, à moins que le message soit consommé par une autre réception équivalente;
- l'équité dans le traitement des communications n'est par contre pas considérée. Un message peut fort bien n'être jamais reçu, même par un processus qui répétitivement effectue des réceptions de ce message, parce qu'il est possible qu'il soit à chaque fois occulté par un message en provenance d'une source différente. De la même manière une requête de réception peut ne jamais être satisfaite, même s'il existe un processus qui répétitivement lui envoie un message, parce qu'il est possible qu'à chaque fois un récepteur concurrent ne consomme ce message. Ceci se produit en particulier lorsque les processus sont composés de plusieurs *threads*.
- si des tampons sont utilisés pour implémenter MPI, alors il doivent être de taille limitée.

Pour contrôler l'évolution des requêtes créées avec les primitives non bloquantes, et permettre un contrôle des communications et de la cohérence des données impliquées, l'interface prévoit une palette assez complète d'opérations.

Outre les fonctions classiques de communication globale comme la diffusion (**MPI_BCAST**) et la barrière de synchronisation (**MPI_BARRIER**), MPI dispose de toutes les opérations de groupe. Ce peut être des opérations purement d'échange de données: la collecte (**MPI_GATHER** et **MPI_GATHERV**) où tous les participants envoient un message à un unique récepteur, l'éclatement (**MPI_SCATTER** et **MPI_SCATTERV**) où un émetteur distribue les données d'un

1. on dit qu'une requête d'émission et une requête de réception **correspondent** lorsqu'elles peuvent être misent en relation l'une avec l'autre, i. e. lorsque l'enveloppe du message en émission satisfait l'enveloppe du message attendu en réception.

message à tous les membres du groupe y compris lui-même, la collecte globale (**MPI_ALLGATHER** et **MPI_ALLGATHERV**) où chaque participant reçoit la concaténation des messages, et la combinaison de l'éclatement et de la collecte (**MPI_ALLTOALL** et **MPI_ALLTOALLV**). Mais ce peut être également des calculs: **MPI_REDUCE** pour une opération standard (addition, multiplication, etc.), et, **MPI_USER_REDUCE** ou **MPI_USER_REDUCEA** pour une opération plus complexe définie par l'utilisateur. Avec ces primitives de calcul le résultat n'est transmis qu'à un seul des participants, mais d'autres modes plus complexes sont possibles: diffuser le résultat (**MPI_ALLREDUCE**, **MPI_USER_ALLREDUCE** et **MPI_USER_ALLREDUCEA**), effectuer l'opération sur un vecteur d'éléments destiné à être éclaté entre les membres du groupe (**MPI_REDUCE_SCATTER**, **MPI_USER_REDUCE_SCATTER** et **MPI_USER_REDUCE_SCATTERA**), ou réaliser un préfixe parallèle (**MPI_SCAN**, **MPI_USER_SCAN** et **MPI_USER_SCANA**). Toutes ces primitives globales, sans exception, font intervenir tous les membres d'un groupe et ne prennent fin que lorsque tous ont contribué à l'achèvement de l'opération.

Enfin, pour une utilisation plus aisée, la conversion des données requise dans les environnements hétérogènes est ici transparente. Pour cela les types des données inclues dans les messages sont contrôlés par MPI grâce à des constructeurs de types qui lui sont propres, et qui sont indépendants de tout langage: **MPI_TYPE_CONTIGUOUS**, **MPI_TYPE_VECTOR**, **MPI_TYPE_HVECTOR**, **MPI_TYPE_INDEXED**, **MPI_TYPE_HINDEXED**, **MPI_TYPE_STRUCT**. La gestion des erreurs et le paramétrage du système de communication sont également accessibles par l'intermédiaire de primitives de l'interface.

2.5 Comparaison des systèmes à diffusion

Après cette étude, non exhaustive, des systèmes à diffusion et environnements de développement et d'exécution, jointe à celles des modèles de programmation parallèle à diffusion du chapitre 1, nous constatons que les choix de réalisation conduisent à des interfaces variées, mais qui possèdent de nombreux points communs. Nous pouvons les comparer par rapport à trois critères: le type de gestion des groupes de processus, les opérations globales qu'ils proposent, et les protocoles de diffusion auxquels ils font appel.

Dans le tableau 2.5 nous constatons en premier lieu que les modèles parallèles basés sur la diffusion, parce que leur objectif est de représenter les architectures de réseaux locaux, sont très peu fournis en opérateurs et n'offrent pas de réelle structure de groupes. Il apparait ensuite clairement que la gestion de groupes de processus est le moyen le plus efficace et le plus utilisé pour réaliser une diffusion sélective (à destination d'un sous-ensemble de processus). Il n'y a toutefois aucun consensus dans la sémantique des groupes, et chaque interface est différente des autres. Cette diversité se retrouve également au niveau des primitives de communication de groupes.

Il en est de même pour les protocoles de diffusion que l'on peut regrouper en trois catégories d'importances égales: ceux qui sont synchrones, les asynchrones avec espace tampon, et les asynchrones sans espace tampon. Isis se distingue des autres systèmes par l'emploi d'un protocole qui garanti un ordre causal; à cheval sur les trois autres, il permet d'obtenir un synchronisme virtuel à partir d'échanges de messages asynchrones. Nous avons classé le protocole de CBS à la fois comme synchrone et asynchrone (sans tampon); en effet, bien qu'il n'y ait pas de rendez-vous entre les participants d'une communication, les opérations d'émission et de réception sont considérées comme simultanées.

	Groupe			Opérations				Protocoles		
	structure	dynamique	recouvrant	1 vers n	n vers 1	n vers n	autres	synchrone	asynchrone	
									sans	tampon
BSP				○ ⊗					✓	✓
CBS	sourdine			*!*				✓		
Isis	fermés	✓	✓	`cbcast`				ordre causal		
Vartalaap	listes	✓	✓	`Vsend / Vreceive`						✓

CCL	ensembles		✓	`bcast`
				`scatter`
					`reduce`
					`gather`
						`combine`
						`concat`
						`index`
							`sync`
							`shift`
							`prefix`
								✓	✓	✓

MPI	ensembles		✓	`BCAST`
				`SCATTER`
					`GATHER`
					`REDUCE`
					`USER_REDUCE`
						`ALLGATHER`
						`ALLTOALL`
						`ALLREDUCE`
						`USER_ALLREDUCE`
						`REDUCE_SCATTER`
						`USER_REDUCE_SCATTER`
							`BARRIER`
							`SCAN`
							`USER_SCAN`
								✓		✓

PVM	listes	✓	✓	`bcast` `mcast`			`barrier`			✓
p4				`broadcast`	`global_op`		`wait_for_end` `global_barrier`			✓

Tableau 2.5 : comparaison des systèmes parallèles à diffusion.

2.6 Vers les systèmes matériels à diffusion: l'exemple du T3D

Le Cray T3D [Cray93,KeSc93] est un exemple de machines parallèles récentes dans lesquelles le système de communication est en grande partie intégré dans le matériel, et de surcroît capable d'effectuer des opérations complexes, telles que les barrières de synchronisation entre autres, qui impliquent un groupe de processeurs. Son architecture comprend un système hôte (système vectoriel traditionnel), des contrôleurs d'entrées/sorties auxquels sont connectés les divers organes périphériques (disques, terminaux, réseau), et un coeur de calcul massivement parallèle. La figure 2.11 illustre l'interconnexion de ces principales composantes de la machine.

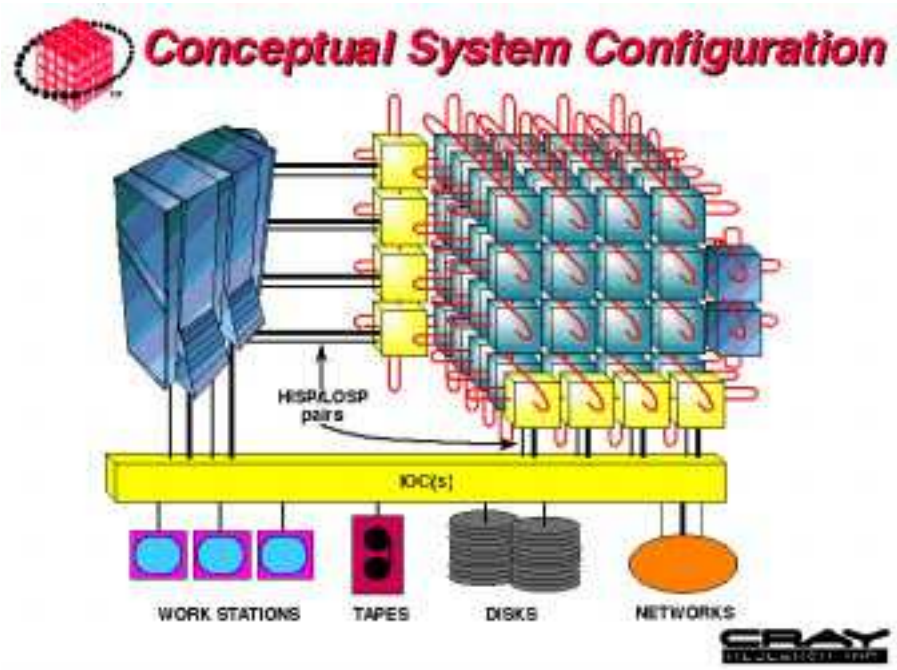


Figure 2.11 : architecture du Cray T3D.

La partie calculateur parallèle, qui nous intéresse ici, se compose de 32 à 2048 noeuds selon la configuration. Comme nous pouvons le remarquer sur la figure 2.11, les noeuds sont interconnectés entre eux par un tore à trois dimensions, et reliés aux autres organes de la machine par des noeuds supplémentaires (cubes plus clairs du premier plan et à gauche du tore) qui servent de passerelles avec les contrôleurs d'entrée/sortie et le système hôte. De plus des noeuds de calcul redondants (cubes sombres à droite du tore) sont ajoutés pour permettre la continuité de l'exécution d'une application en cas de pannes des noeuds du tore.

Dans le tore du T3D tout noeud physique est constitué de deux éléments de calcul, d'un contrôleur asynchrone d'accès direct à la mémoire (contrôleur mémoire), d'une interface d'accès au réseau et d'un routeur de messages. Notre intention n'étant pas ici de présenter en détails l'architecture du T3D, nous ne présenterons ni particularités des passerelles d'entrées/sorties, ni le système d'accès non uniforme à la mémoire.

Les deux éléments de calcul d'un noeud se composent chacun d'un processeur (DEC Alpha 21064), d'une mémoire locale et d'un ensemble de circuits qui étendent les fonctionnalités de contrôle et d'adressage du micro-processeur. Par ailleurs une horloge centrale fournit à l'ensemble des noeuds un signal toutes les 6,67 ns. La figure 2.12 représente les différents composants d'un noeud.

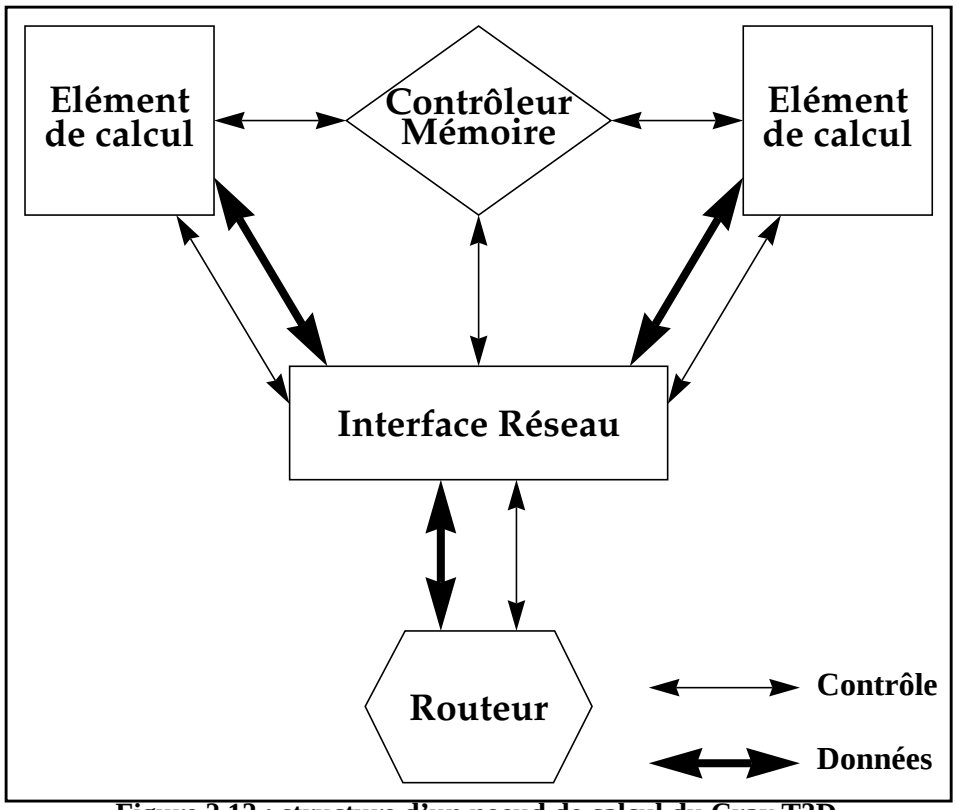


Figure 2.12 : structure d'un noeud de calcul du Cray T3D.

Un routeur est relié avec deux autres sur chacun des trois axes de manière à former les trois dimensions du tore. Deux routeurs sont connectés par un lien de communication bidirectionnel qui comprend deux canaux unidirectionnels, un pour chaque sens de communication. Ces canaux, comme cela est représenté sur la figure 2.13, comportent une voie d'échange de données, un ensemble de signaux de contrôle pour l'identification d'une «requête» ou d'une «réponse» et du tampon de réception, et, une voie d'acquittement qui permet d'informer l'émetteur que le tampon est ou non plein.

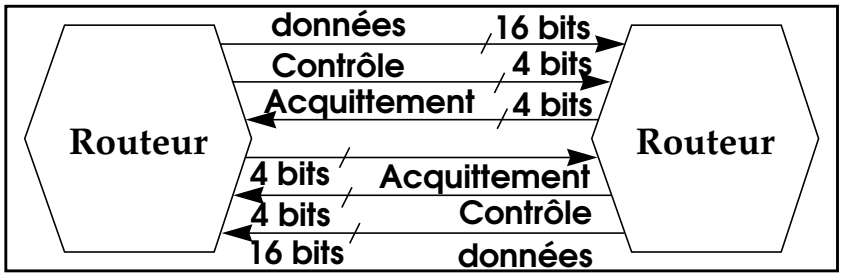


Figure 2.13 : connexion entre deux routeurs dans le Cray T3D.

Comme les noeuds sont logiquement numérotés sur chaque dimension, le routage mis en oeuvre par les routeurs se fait très simplement dans l'ordre des dimensions. A ce niveau le routage tolère les pannes de liens en permettant aux messages de circuler dans l'un ou l'autre des sens d'une dimension.

Pour prévenir les interblocages le système d'échange de messages du T3D dispose de quatre canaux virtuels de communication qui définissent quatre classes de tampons: deux sont alloués à des messages de type «réponse», et les deux autres à des messages de type «requête». La dis-

tinction entre tampons «requête» et tampons «réponse» permet de ne pas provoquer d'interblocages entre des noeuds voisins qui essaieraient de se transmettre mutuellement des messages.

Pour éviter les interblocages qui pourraient se produire entre les noeuds sur une dimension, deux tampons sont alloués pour chacun de ces types («requête» et «réponse»); la règle d'utilisation associée à ces deux tampons définit un lien particulier dans la dimension qui, s'il doit être emprunté implique l'emploi de l'un des tampons, et s'il ne l'est pas de l'autre tampon.

En plus des routeurs, le système de communication du T3D comprend des interfaces d'accès au réseau, des circuits d'échange de messages et des circuits qui gère des opérations de synchronisation. Un circuit d'échange de message, avec l'aide de l'interface d'accès au réseau qui assemble et désassemble les messages, décharge le processeur auquel il est rattaché des tâches de réception et d'expédition de messages. Il gère également les accusés de réception, et les messages de rejet lorsque les tampons de réception sont pleins et qu'un message ne peut être reçu. Enfin ce circuit, en conjonction avec l'interface d'accès au réseau, détecte les erreurs issues du réseau: données corrompues ou messages perdus.

Le circuit associé aux opérations de synchronisation permet de manipuler deux types d'outils: les barrières de synchronisation et les «eurekas». Les premiers définissent des points dans un programme qui impliquent la synchronisation de tous les processeurs associés à la barrière. Pour cela ces circuits disposent de deux registres de huit bits où chaque bit correspond à un outil de synchronisation. Dans le cas d'une barrière de synchronisation, ce bit est connecté avec ses homologues des différents circuits par une arborescence de portes «et» dont le résultat est retourné à l'ensemble des processeurs. La figure 2.14 présente un exemple simplifié, avec huit processeurs (P1 à P8), d'une telle arborescence.

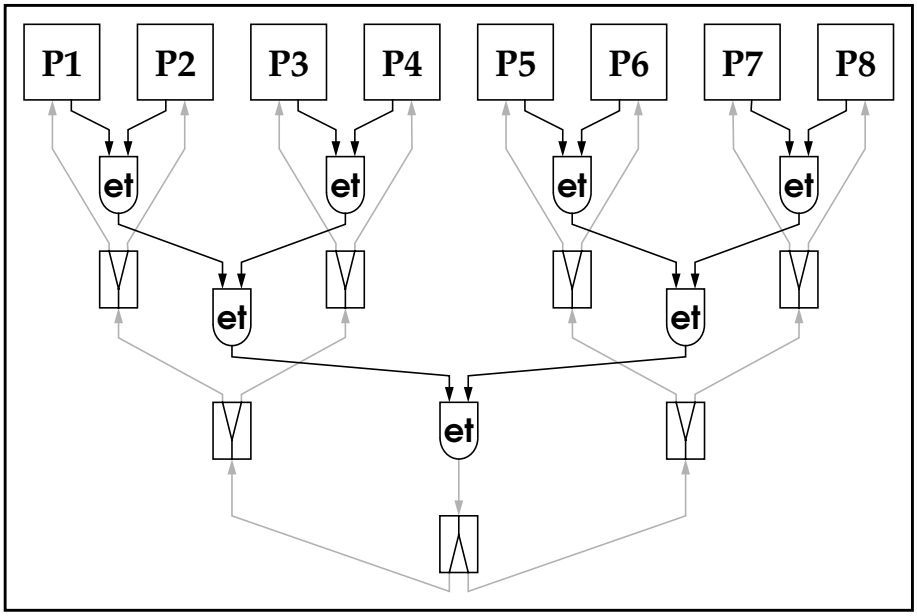


Figure 2.14 : Circuits de réalisation d'une barrière de synchronisation du T3D.

La synchronisation est obtenue lorsque tous les processeurs concernés par la barrière ont positionné le bit du registre à 1; ils en sont avertis soit par attente active, soit par interruption.

Les «eurekas» servent à avertir l'ensemble des composantes d'un programme que l'une d'entre elles à atteint un point particulier de son exécution. Pour une recherche en parallèle dans une base de données par exemple, cela permet d'arrêter le programme dès que la ou les données désirées ont été trouvées dans l'un des processeurs, plutôt que d'attendre que tous aient effectué leur travail. Les «eurekas» sont générés en changeant les portes «et» en portes «ou» dans l'arborescence des barrières de synchronisation; c'est-à-dire qu'ils fonctionnement comme un «ou» logique global.

Le matériel du T3D permet d'une part d'associer à chaque bit des registres l'un ou l'autre de ces fonctionnements. Il inclut d'autre part la possibilité d'isoler les uns des autres plusieurs sous-ensembles de processeurs vis-à-vis d'un bit des registres.

2.7 Conclusion

Ce chapitre donne une image assez fidèle des résultats des travaux de recherche sur la diffusion et les communications globales; c'est-à-dire que peu de langages et systèmes d'exploitation offrent un tel outil d'échange de messages. En revanche, la quasi-totalité des environnements de développement et d'exécution parallèle ont adopté ce type de communications, certains même lui sont spécifiquement dédiés.

Fonctionnalité de base aux communications globales, la diffusion a donc pris ces dernières années une importance croissante. Par contre, si beaucoup de travaux ont été menés sur la diffusion de bas niveau, i. e. le routage de messages à diffusion dans un réseau d'interconnexion, ou sur les protocoles et primitives offertes aux utilisateurs, bien peu ont réuni ces deux aspects. Notre thèse s'inscrit dans ce cadre et propose à la fois une solution au routage des messages à diffusion dans un réseau d'interconnexion quelconque, et une machine virtuelle à diffusion qui donne aux utilisateurs les protocoles et primitives nécessaires à son application.

Par ailleurs, l'émergence de nouvelles architectures de machines, qui disposent de matériels performants et dédiés aux communications, rend nécessaire d'avoir des algorithmes de routage et d'acheminement de messages aptes à être intégrés dans des circuits. Nous avons donc pris en compte cette contrainte dans l'élaboration de nos solutions pour l'acheminement de messages à diffusion pour des réseaux d'interconnexions à topologies quelconques.

Partie II : Conception d'un Routeur à Diffusion

La construction d'une machine virtuelle à diffusion pourrait se satisfaire de tout système de communication point-à-point pour construire des protocoles de diffusion: puisqu'au-dessus de celui-ci le réseau de processeurs est virtuellement une machine totalement connectée; la diffusion se résumerait à envoyer un message vers chacun des processeurs concernés. Seulement, ne serait-ce que du point de vue des performances, ce type de communications nécessite plus que la seule fonction d'expédition de messages vers n'importe quelle destination, et requiert un support adapté, pour le contrôle ou la synchronisation associés à la diffusion par exemple. Si avec peu de processus ce procédé «simpliste» peut ne pas s'avérer trop pénalisant, dans le contexte du parallélisme massif cela n'est pas acceptable.

Il faut donc, avant de pouvoir bâtir une machine virtuelle à diffusion réellement utilisable, faire évoluer un routeur point-à-point afin qu'il puisse offrir un service supplémentaire de diffusion de messages. Ce service doit être efficace et indépendant de l'architecture de la machine et de la topologie du réseau d'interconnexion. Il s'agit plus précisément, et c'est là le sujet de cette deuxième partie, de proposer un routeur apte aussi bien aux échanges de messages point-à-point qu'aux divers protocoles de diffusion de messages.

Notre objectif étant de faire cohabiter ces deux schémas de communication au sein d'un même routeur, nous commencerons cette partie par la présentation des principales méthodes de routage et d'acheminement (pour plus de détails voir par exemple [MMS90], [Gonz91] et [Mug93]). Nous nous intéresserons plus particulièrement au routeur de ParX développé par notre équipe. Puis nous étudierons les différentes méthodes de diffusion qui ont été proposées, nous les comparerons et évaluerons leur efficacité. Enfin nous proposerons une solution nouvelle à ce problème, qui répond aux exigences du parallélisme et présente de surcroît des caractéristiques intéressantes vis-à-vis des fonctions de routage point-à-point.

Chapitre 3 : Routage et Acheminement Point-à-point

Comme nous l'avons vu tout au long de la première partie de cette thèse, la communication par échanges de messages est un élément primordial du parallélisme. Plus que d'être un modèle de programmation, c'est aussi le seul paradigme à partir duquel peuvent se concevoir les architectures massivement parallèles flexibles, extensibles, «scalables». Mais s'il rend possible la construction de machines à haut degré de parallélisme, il introduit un problème délicat et crucial: celui de l'acheminement **correct** des messages dans le réseau d'interconnexion, fonction qui ne peut être laissée simplement à la charge des applications.

La situation la plus simple pour que deux processeurs puissent s'échanger des messages est celle où ils sont directement connectés l'un à l'autre par un lien de communication; il n'y a alors plus à se soucier du routage. Comme nous l'avons déjà fait remarquer, les réseaux complets qui rendent les noeuds voisins deux à deux ne peuvent être utilisés que pour un nombre très limité de processeurs, de sorte que les machines parallèles emploient des réseaux dans lesquels il faut effectuer un routage des messages. Il est possible d'utiliser un réseau dynamiquement reconfigurable pour réaliser, à la demande, des connexions entre processeurs. Mais encore faut-il se placer dans l'hypothèse supplémentaire d'un réseau non bloquant, c'est-à-dire où l'établissement d'une nouvelle connexion puisse se faire indépendamment de celles déjà existantes.

Pour obtenir des réseaux qui répondent à cette condition on utilise un ensemble de commutateurs qui relient toutes les entrées du circuit à tous ses sorties. Ces circuits, appelés ***crossbars***, peuvent ainsi connecter *n* entrées à *p* sorties comme cela est représenté dans la figure 3.1.

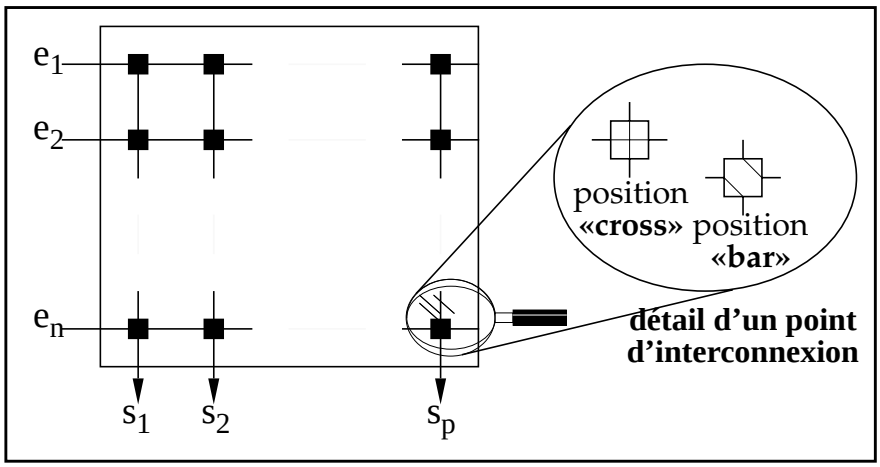


Figure 3.1 : structure d'un réseau de crossbars.

Il reste néanmoins à s'assurer que, pour chaque nouvelle connexion, l'entrée et la sortie désirées sont effectivement libres. De plus les contraintes technologiques d'un tel dispositif de commutation limite son utilisation aux machines de petites tailles (de l'ordre de la centaine de connexions). Les machines plus importantes sont construites à partir de réseaux multi-étages (*Clos*, *Benès*, etc.) [Clos53, Benes62]. Ces réseaux emploient des crossbars de tailles restreintes et les arrangent pour obtenir l'interconnexion d'un grand nombre d'éléments. Ils peuvent également fournir d'autres propriétés comme le caractère non bloquant, ou la limitation du nombre de transistors pour réduire le coût de fabrication.

Les études menées sur la configuration ou reconfiguration de ces architectures, dont une synthèse se trouve dans [Mug93], montrent que les mécanismes induits, logiciels et matériels, sont trop coûteux pour être mis en oeuvre à chaque transfert d'un message, surtout lorsqu'il s'agit de programmes fortement communicants. L'utilisation de cette technique reste donc limitée à la configuration du réseau en début d'exécution des applications, et, plus rarement, à un changement de topologie d'interconnexion entre différentes phases d'un programme. En somme il s'avère que le routage des messages est inévitable dans les machines extensibles.

Dans ce chapitre nous traiterons de ce problème dans le cas de communications point-à-point. Pour ce faire nous commencerons par présenter les techniques d'acheminement les plus fréquentes dans les machines parallèles. Puis nous exposerons en quoi consiste une fonction de routage, et quels doivent être ses caractéristiques pour garantir la correction du système de communication. Nous aborderons ensuite les principales méthodes de construction de fonctions de routage correctes, essentiellement celles qui sont issues des travaux de recherche de notre équipe, et, qui sont utilisées dans le routeur de notre noyau de système ParX. Nous conclurons ce chapitre en étudiant les possibilités qu'offre le routage dit par intervalles, et notamment en ce qui concerne l'implémentation de fonctions de routage dans un routeur matériel.

3.1 Techniques d'acheminement de messages

Choisir une technique d'acheminement c'est choisir les traitements appliqués en chaque noeud du réseau pour que tout message parvienne à destination, et si possible de manière optimale. La technique d'acheminement la plus ancienne, connue sous le nom de **commutation de circuits**, est issue de la téléphonie et consiste à réserver, avant que le transfert de données ne commence, un chemin entre les deux protagonistes d'une communication.

Selon ce principe un signal de réservation est en premier lieu envoyé vers la destination, et au fur et à mesure qu'il traverse le réseau il réserve les liens qu'il emprunte, jusqu'à atteindre le dernier noeud; un acquittement est alors retourné à l'expéditeur. Après la réception de cet acquittement la source peut transmettre son message, puis, ceci fait, libérer les ressources allouées. La réservation des ressources est le principal facteur du taux particulièrement faible d'occupation des liens de communication, ce qui fait singulièrement chuter les performances globales du système.

Pour remédier à ce problème la commutation ne doit pas être opérée sur les ressources mais entre les messages eux-même. Avec cette règle les liens sont alloués au cours de la progression du message, et libérés immédiatement après qu'il les ait empruntés. Les données sont donc acheminées pas à pas, de noeud en noeud. Cette façon de faire nécessite évidemment de mémoriser les messages dans chaque noeud, ce qui lui vaut son nom de **«store-and-forward»**. Certes les liens sont beaucoup mieux exploités, mais il faut cette fois tenir compte du retard introduit, en chaque noeud, par la mémorisation des données; retard inutile lorsque le lien suivant dans le chemin n'est pas occupé.

Afin d'atténuer ce phénomène le message peut être découpé en plusieurs paquets, et profiter ainsi d'un «*effet pipe-line*» ou de l'existence de plusieurs chemins. Le gain obtenu doit toutefois être nuancé par les temps de découpage et de réassemblage des messages qui s'avèrent des facteurs importants dans les performances globales d'un système. L'avantage principal de cette méthode est qu'elle permet de borner l'espace mémoire nécessaire en chaque noeud pour stocker les paquets en transit.

Une solution réelle à l'oisiveté des liens de communication consiste à ne pas effectuer de stockage des messages ou des paquets: lorsque le lien suivant à emprunter a été déterminé, et si ce lien est libre, la retransmission commence immédiatement et se fait au fur et à mesure que le message est reçu. Cette technique, qui porte le nom de **«wormhole»**, permet ainsi d'obtenir à la fois un délai de transmission identique à celui de la commutation de circuit, et un taux d'utilisation des liens nettement supérieur. Ces améliorations, très nettes si le réseau est peu chargé et que la probabilité de trouver un lien libre est grande, sont malgré tout pondérées lorsqu'un lien qui n'est pas disponible stoppe la progression du message. En effet dans ce cas, les données sont mémorisées dans l'ensemble des ressources qu'elles occupent, ressources qui restent alors inutilisables jusqu'à ce que le message puisse de nouveau continuer sa progression. Une telle situation est schématisée dans la figure 3.2 ci-dessous.

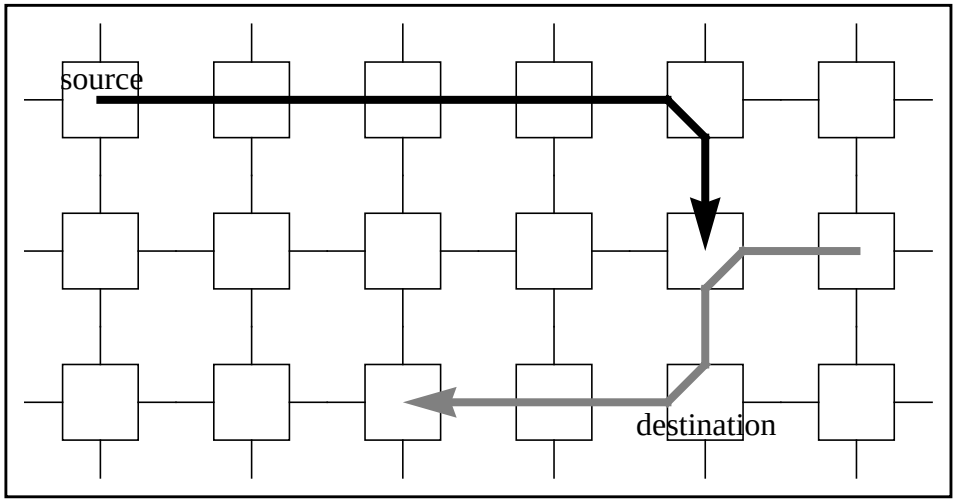


Figure 3.2 : occupation des ressources pour le wormhole.

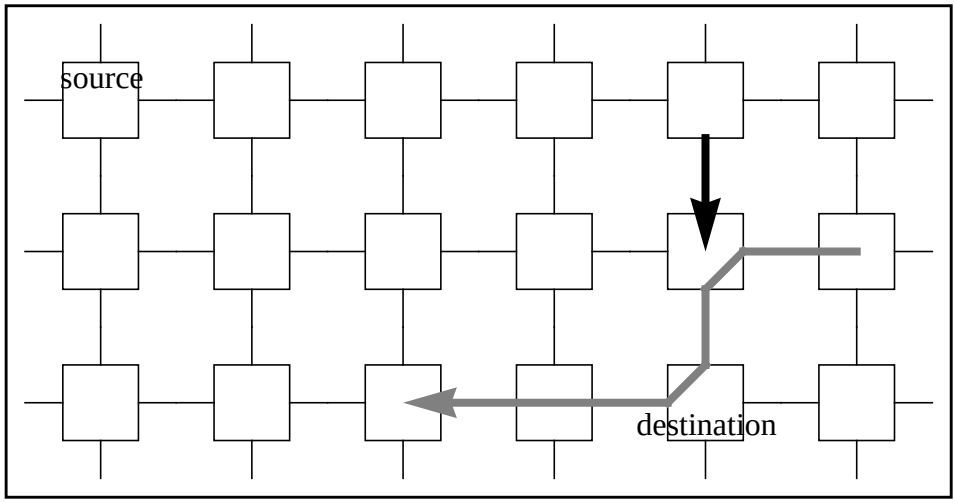


Figure 3.3 : occupation des ressources du virtual cut-through.

Le problème vient du fait qu'avec le wormhole les noeuds ne sont capables que de stocker une «infime» partie du message (quelques bits). Dans le but de limiter le nombre de ressources inefficacement bloquées, il est possible d'utiliser la technique du **«virtual cut-through»** et de doter les noeuds d'une véritable capacité de stockage. Ainsi lorsque la tête du message se trouve bloquée, le reste du message pourra continuer d'avancer pour être mémorisé dans le dernier noeud, libérant du même coup des ressources jusqu'alors réservées. C'est ce que nous pouvons remarquer dans la figure 3.3, où, dans la même situation que précédemment (cf. figure 3.2), seuls un tampon de la taille du message et un lien de communication sont occupés par le message en attente.

Le choix de l'une ou l'autre de ces techniques dépend du cadre d'utilisation. Alors que le store-and-forward est la seule méthode réellement utilisable pour réaliser un routeur logiciel, le choix entre le wormhole et le virtual cut-through pour un routeur matériel est question de performances et de complexité: si le wormhole s'implante plus aisément dans le matériel parce qu'il ne nécessite que peu d'espace de stockage, le virtual cut-through devrait être plus performant lorsque le réseau est fortement chargé.

3.2 Fonction de routage

Dans le routage la technique d'acheminement détermine comment les messages traversent les noeuds, mais elle ne définit pas comment les liens à emprunter sont sélectionnés. Cette décision revient à la fonction de routage. Une telle fonction consiste en fait à calculer un chemin pour toute paire de processeurs. Pour la plupart des réseaux réguliers cette fonction est simple, comme dans le cas du routage **«e-cube»** qui est à la fois rapide et facile à mettre en oeuvre sur un hypercube. De telles fonctions sont d'ailleurs souvent implanter dans de petits opérateurs câblés.

Pour des réseaux de topologies quelconques et irrégulières, il s'agit en fait d'un précalcul dont le résultat est utilisé à chaque transfert de données, ce qui implique d'ailleurs un stockage de ces informations de routage. Une première façon d'opérer est de mémoriser en chaque noeud tous les chemins dont il est la source, et coder dans chaque message le chemin qu'il doit suivre. Cependant, d'une part cette information est dépendante de la taille du réseau, et d'autre part cela se traduit par une perte notable de la bande passante du réseau. En effet le trafic alors généré par les en-têtes de messages rend cette méthode inadaptée aux réseaux de grande taille. C'est pourquoi les systèmes d'échanges de messages distribuent l'information de routage.

3.2.1 Représentations de la fonction de routage

Soit $G = (P,L)$ le graphe d'interconnexion physique des processeurs, où P est l'ensemble des noeuds, L l'ensemble des liens de communication et B l'ensemble des booléens. Une fonction de routage peut se définir formellement comme une application:

$$f: L \times P \times P \times L \rightarrow B$$
$$(e, n, d, s) \mapsto vrai \vee faux$$

où $f(e,n,d,s)$ est vrai si un message reçu au noeud n, par le lien e et à destination du processeur d, peut être retransmis sur le lien s. On notera qu'il est ainsi possible d'avoir plusieurs liens de sortie valides pour une même destination et un même lien d'arrivée.

Avant d'aller plus loin dans la représentation d'une fonction de routage, il est important d'introduire les différentes caractéristiques qui peuvent lui être attribuées:

- *déterministe*: f est dite déterministe si elle définit un unique chemin entre toute paire de processeurs;
- *adaptative*: f est adaptative si elle définit plusieurs chemins entre les processeurs, et permet alors de prendre en compte des conditions dynamiques telles que la panne ou la congestion d'un lien;
- *valide*: f est valide si dans aucun des chemins qu'elle définit il n'existe de boucle infinie. Une condition suffisante pour obtenir cette caractéristique est de ne pas utiliser plus d'une fois un lien dans un chemin;
- *minimale*: f est dite minimale si elle définit le plus court chemin entre chaque paire de noeuds.

Pour distribuer cette information de routage, chaque noeud n reçoit une restriction f_n de f:

$$f_n: L \times P \times L \rightarrow B$$
$$(e, d, s) \mapsto f(e, n, d, s)$$

Cette restriction est encodée en chaque noeud dans une table de routage R, sous forme d'un tableau à deux dimensions:

$$R_n[e, d] = \{ s \in L, f_n(e, d, s) = vrai \}$$

Cette représentation des tables de routage est dite directe, car l'ensemble des liens de sortie est directement accessible à partir du lien d'arrivée et de la destination. Pour des fonctions déterministes, la représentation interne de R_n est juste une matrice de numéros de liens. Par contre avec des fonctions adaptatives chaque entrée dans cette table est un vecteur de bits (un bit par lien de sortie).

Il est également possible d'accéder à l'information de routage, non plus à partir du lien d'entrée et de la destination, mais par rapport aux liens d'entrée et de sortie. Dans ce cas la table de routage est inversée, et chaque entrée détermine l'ensemble des destinations (mémorisées dans un vecteur de bits) associées à une paire de liens d'entrée et de sortie. En tout noeud n, une telle table est donc définit comme suit:

$$R_n^{-1}[e, s] = \{ d \in P, f_n(e, d, s) = vrai \}$$

La plupart des réalisations de tables inversées font abstraction du lien d'entrée; la table de routage se résume alors à un simple vecteur, où chaque lien de sortie se voit attribué l'ensemble des destinations auxquelles il mène:

$$R'^{-1}_n[s] = \{ d \in P, \forall e \in P, f_n(e, d, s) = vrai \}$$

3.2.2 Correction de la fonction de routage

Quelles que soient la technique d'acheminement, la fonction de routage et sa représentation interne retenues, il y a concurrence d'accès et partage des mémoires tampons et des liens de communication liés au routage. De ce fait il est indispensable de garantir la correction du système de communication, correction qui impose au moins les propriétés suivantes: ***validité***, ***absence d'interblocage***, ***absence de famine*** et ***équité***. D'autres propriétés comme l'équilibrage de la charge des liens peuvent être ajoutées à cette liste bien que cela ne soit pas essentiel à la correction du routage; la tolérance aux pannes doit aussi être considérée dans les architectures parallèles, bien que l'hypothèse d'absence de pannes est souvent faite.

La validité est évidemment la propriété la plus fondamentale, car elle impose que tout message atteigne sa destination et ne reste pas continuellement dans le système de routage. Sans autre contrainte, il suffit de trouver simplement un chemin, quel qu'il soit, entre toute paire de processeurs. Cette garantie ne prend réellement son sens qu'avec l'absence d'interblocage; en effet puisque, comme nous le verrons, cette seconde propriété restreint le nombre de chemins possibles, il est important de s'assurer que tout message arrivera bien à destination.

En fait l'interblocage est le principal problème du routage. Il se caractérise par un ensemble de messages bloqués, qui attendent qu'au moins une des ressources, lien ou tampon, soit libérée par l'un des membres de l'ensemble. Il y a trente ans déjà E. W. Dijkstra caractérisait cette situation par l'existence d'un cycle d'inter-dépendance entre les messages bloqués [Dijk68].

On peut distinguer différents cas selon l'origine de l'interblocage, mais nous ne nous intéresserons ici qu'à ceux qui sont sous la responsabilité du routage. A ce niveau le cas le plus simple se produit entre deux processeurs voisins (interblocage local), chacun ayant ses tampons remplis de messages en direction de l'autre (figure 3.4a). Cette situation se résout facilement en dotant les noeuds de deux classes de tampons, l'une en réception et l'autre en émission (figure 3.4b).

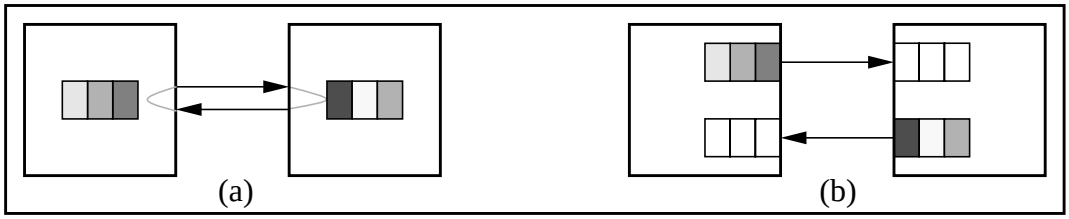


Figure 3.4 : résolution d'un interblocage local.

Mais cette précaution ne suffit plus lorsqu'il se forme un cycle de dépendances de plus de deux noeuds. Cette situation est due à la fonction de routage qui définit les chemins empruntés par les messages, et dans ces chemins les dépendances entre les liens et les tampons. En d'autres termes, à toute fonction de routage correspond un graphe de dépendances associé qui modélise l'utilisation des ressources dans tous les chemins [DalSei87]. Il s'agit d'un graphe orienté qui se construit de la manière suivante:

- les sommets du graphe de dépendances sont les liens du réseau;
- deux sommets l_1 et l_2 sont reliés par un arc (l_1,l_2) si, et seulement si, il existe un chemin induit par f dans lequel l_1 est un prédécesseur direct de l_2.

Par rapport à ce graphe, un interblocage se produit lorsque les deux conditions suivantes sont remplies:

1- le graphe de dépendances comporte des cycles;

2- les tampons sont pleins dans l'un des cycles.

Pour éviter qu'un interblocage ne se produise il peut suffire d'utiliser des tampons de taille suffisamment grande dans le routeur, ou d'un nombre suffisant de classes de tampons, ainsi que des règles d'utilisation de ces classes afin que les tampons ne soient jamais pleins. C'est par exemple ce qui est fait dans le routeur VCR[1] [DHN90, DHN91]. Mais dans bien des cas, puisqu'il n'est pas envisageable de connaître les besoins exacts en échanges de messages d'une application, ces techniques sont restreintes à des réseaux d'interconnexion particuliers, ou à un nombre fixe de canaux de communication.

D'autres techniques de prévention basées sur ce principe, dont le lecteur trouvera une présentation plus complète dans [Mug93], opèrent un contrôle à l'exécution pour s'assurer que tout message disposera des ressources nécessaires pour être acheminé à sa destination. Ces solutions font donc intervenir une politique d'admission des messages dans la couche de routage. Le délai d'admission est alors augmenté, et c'est justement ce temps, parce qu'il est le facteur de performances le plus sensible, que tout système de communication cherche à réduire.

Une autre méthode, dite de **détection-guérison**, qui n'impose aucune contrainte à la fonction de routage, consiste à activer au cours de l'exécution un mécanisme particulier d'élimination des interblocages lorsque ceux-ci sont décelés. Mais, ne serait-ce que pour des raisons de coût, les techniques de détection-guérison sont particulièrement mal adaptées au parallélisme.

Finalement, pour concilier absence d'interblocage, faisabilité et moindre coût en termes de performances, la solution à ce problème est d'apporter des restrictions à la fonction de routage. Ces contraintes ont pour but de produire une fonction pour laquelle le graphe de dépendances ne comporte pas de cycle. Certes ceci se fait généralement au détriment de la longueur des chemins, mais plus aucun contrôle n'est à effectuer pendant l'exécution, et les tampons peuvent ainsi être de taille bornée et de surcroît de taille indépendante du réseau.

Les deux dernières propriétés, famine et équité, sont liées si l'on veut obtenir un routage correct des messages. La famine se manifeste lorsqu'un message ne peut accéder à une ressource (lien ou tampon), alors que celle-ci se libère de temps à autre. L'équité quant à elle vise à ne pas privilégier certains messages par rapport à d'autres, de manière à ne pas introduire de famine.

L'équité peut concerner tous les messages sans exception, mais elle s'applique dans bon nombre de réalisations avec différents niveaux de priorité. Quoiqu'il en soit l'absence de famine et l'équité se résolvent aisément en employant des politiques de service de type *premier entrée/ premier sorti* par exemple.

1. Virtual Channel Router (routeur à canaux virtuels).

3.3 Fonctions de routage correctes

Selon les critères que nous avons dégagés dans la section précédente, une fonction de routage, pour être correcte, doit être valide, et son graphe de dépendances ne doit pas contenir de cycle pour garantir l'absence d'interblocages. La minimisation du temps de transfert et l'encombrement du réseau peuvent également être pris en considération dans l'élaboration d'une fonction de routage. Ainsi on pourra aboutir à une fonction adaptative ou une fonction sur les plus courts chemins.

La recherche de solutions au routage correct a été centrée sur les topologies les plus fréquentes dans les architectures parallèles. Cette démarche a aboutie à des solutions optimales pour les réseaux réguliers comme les grilles ou les hypercubes. Par exemple dans une grille il existe au moins un chemin minimal entre toute paire de processeurs. Pour des processeurs situés sur la même ligne (resp. colonne) il suffit de «router» sur la ligne (resp. colonne). Pour les autres paires le routage peut se faire en premier lieu sur la ligne, puis sur la colonne; symétriquement un routage sur la colonne puis sur la ligne est équivalent, vis-à-vis de la longueur du chemin.

Les grilles et les hypercubes sont des cas typiques de réseaux pour lesquels une fonction de routage optimale et correcte existe, qui porte en l'occurrence le nom d'**«e-cube routing»**. L'inconvénient du e-cube routing est qu'il n'est valable que pour les classes de réseaux pour lesquelles il a été défini. De plus, bien qu'il existe des algorithmes adaptatifs, le routage e-cube traditionnel devient invalide si une panne survient (d'un lien ou d'un noeud).

Dans le cas général il s'agit de trouver un graphe qui recouvre la topologie d'interconnexion, dans lequel le routage sera effectué et où aucun interblocage ne peut se produire. Dans cet esprit, et puisque le réseau est connexe, il est toujours possible de construire un arbre recouvrant. L'arbre étant un graphe acyclique, le routage sur une telle topologie est donc sans interblocage (cf. figure 3.5a).

De même un chemin hamiltonien (cf. figure 3.5b) garantit l'absence d'interblocage en n'introduisant aucune dépendance cyclique entre les liens. Par contre il n'existe pas toujours de chemin hamiltonien: une condition suffisante pour qu'un réseau $G=(P,L)$ possède au moins un chemin hamiltonien est qu'il soit un graphe simple d'ordre n tel que toute paire x,y de sommets non adjacents vérifie $d_G(x) + d_G(y) \geq n$ (où $d_G(x)$ est le degré de x).

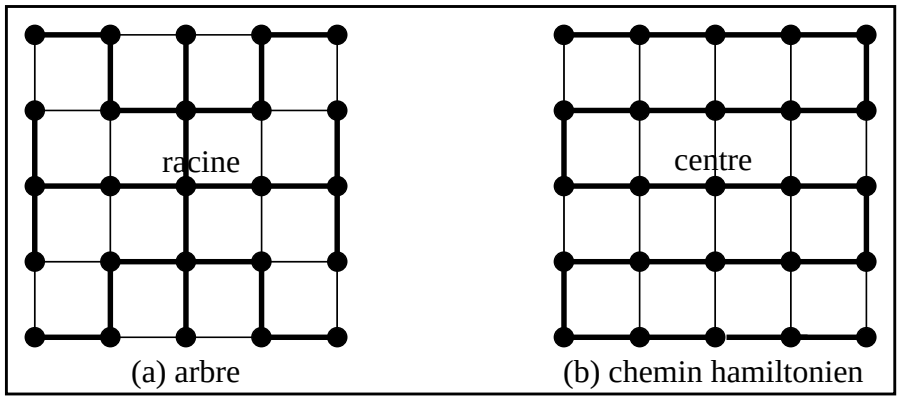


Figure 3.5 : exemples de graphes recouvrants.

L'inconvénient de ces solutions est qu'elles n'exploitent pas complètement le réseau d'interconnexion, qu'elles laissent beaucoup de liens inutilisés. Une première conséquence est que le diamètre se trouve considérablement augmenté: il est au mieux le double du diamètre original

dans le cas de l'arbre, et égal au nombre de noeuds pour le chemin hamiltonien. Mais surtout cela conduit à un ensemble de chemins bien peu et même mal réparti dans le réseau, en faisant du centre du chemin hamiltonien et de la racine de l'arbre des points de trafic potentiellement très élevé, en d'autres termes des goulots d'étranglement.

Traian Muntean qui dirige notre groupe de recherche a démarré dès 1987/88 un projet pour remédier à ces problèmes. Ces travaux ont montrés qu'il est possible de définir des règles qui régissent l'exploitation dans les chemins des liens hors du graphe recouvrant, et ainsi de se servir de ce graphe pour profiter de sa propriété d'absence d'interblocage. C'est ce qui est proposé par Néstor González pour l'arbre [Gonz91]; il montre qu'avec un arbre, construit en largeur d'abord, les trois règles suivantes suffisent:

1- un lien descendant ne doit pas précéder un lien ascendant;

2- un lien horizontal ne doit pas précéder un lien ascendant;

3- les cycles de liens horizontaux dépendants sont interdits.

où le niveau d'un noeud correspond à la distance qui le sépare de la racine, et où un lien est dit ascendant s'il est emprunté depuis un noeud de niveau *i* vers un noeud de niveau *i*-1, descendant s'il est emprunté dans le sens inverse, et horizontal entre des noeuds de même niveau. On obtient alors une fonction de routage toujours exempte d'interblocage, mais mieux distribuée à travers le réseau tout entier et de diamètre plus modéré.

Si l'on considère l'exemple du tore 4x4 (cf. figure 3.6a), l'arbre calculé pourrait être celui de la figure 3.6b. Dans cet arbre le chemin entre les processeurs 14 et 15 emprunterait les noeuds intermédiaires 2 et 3; or ces processeurs sont voisins, ils peuvent donc, sans enfreindre les règles précédentes, et en utilisant deux classes de tampons pour la réception et l'émission, communiquer directement par leur lien commun. Toujours sur cet exemple, pour améliorer le chemin dans l'arbre entre les processeurs 9 et 7 (9, 5, 1, 2, 3 puis 7) il n'est pas possible d'emprunter les noeuds 9, 12 puis 11 car la règle 1 n'est alors pas respectée. Par contre le chemin qui passe par les processeurs 5 et 8 est valide.

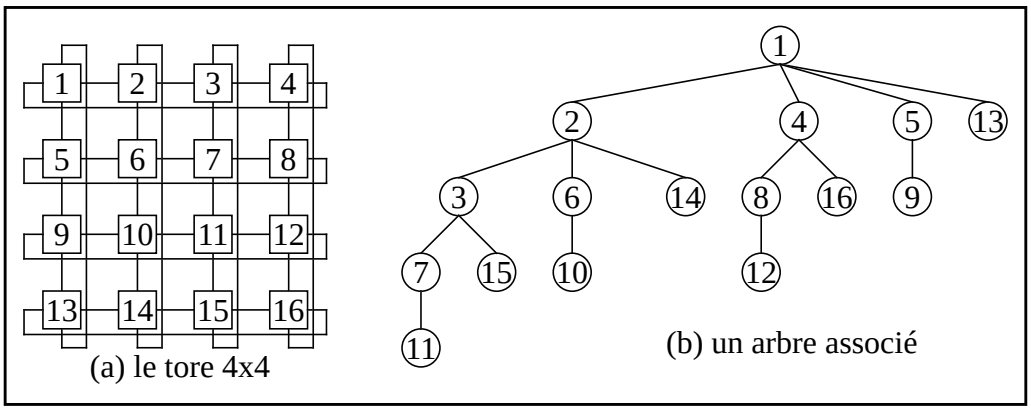


Figure 3.6 : le tore 4x4 et un de ses arbres recouvrants.

La méthode présentée par Léon Mugwaneza dans sa thèse [Mug93], construit une fonction de routage à partir d'un cycle eulérien. Lorsque le réseau n'admet pas un tel cycle, soit lorsqu'il existe des noeuds de degré impair, il suffit de multiplexer les liens en leur associant deux classes de tampons indépendantes pour d'obtenir un réseau où tous les noeuds sont de degré pair et où il existe un cycle eulérien. Après sa construction, le cycle est amputé d'un lien et, muni d'un

sens afin de numéroter les liens et de distinguer les liens empruntés dans le sens du cycle ou liens directs, de ceux traversés en sens inverse ou liens indirects. A partir de cette orientation, L. Mugwaneza fixe les règles suivantes qui permettent de calculer une fonction de routage correcte:

1- un lien direct ne doit pas précéder un lien indirect;

2- un lien direct ne doit pas précéder un lien direct de numéro inférieur;

3- un lien indirect ne doit pas précéder un lien indirect de numéro supérieur.

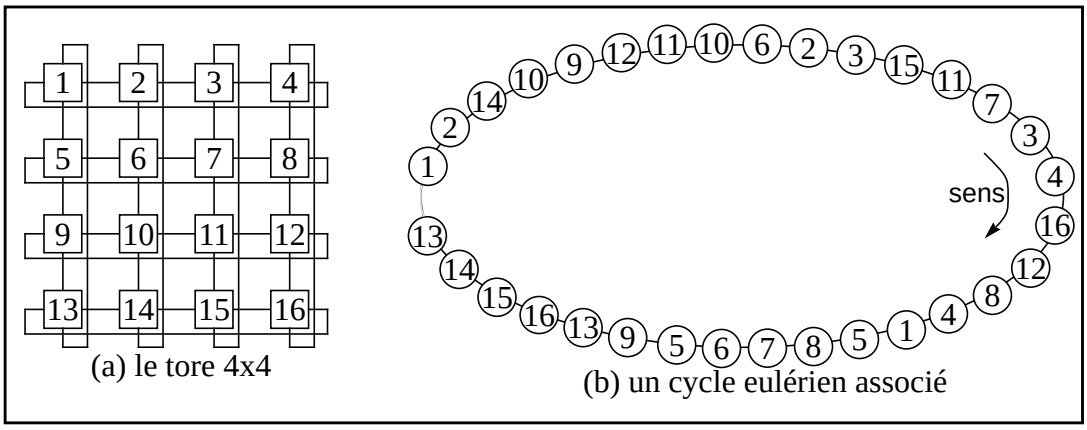


Figure 3.7 : le tore 4x4 et un de ses cycles eulériens.

Comme nous le voyons sur la figure 3.7, cette méthode a l'avantage de conserver le voisinage des noeuds dans le graphe de routage; et ainsi de permettre la communication entre processeurs voisins par leur lien commun. Sur cet exemple, les routes 9-12-16, 16-12-11 ou 11-12-8 ne sont pas permises: la première car elle transgresse la règle 1, la seconde parce qu'elle ne respecte pas la règle 2, et la dernière en vertu de la règle 3. Par contre les routes 9-5-6-7 ou 9-5-8-7 sont autorisées.

3.4 Routage par intervalles

Excepté pour le routage de type «e-cube» pour lequel une simple fonction arithmétique est suffisante, l'information nécessaire au routage est représentée dans des tables telles que nous les avons présentées à la section 3.2.1. Quelle que soit la forme de la table que chaque noeud acceuille, sa taille est dépendante du nombre de noeuds, c'est-à-dire de la taille du réseau. Si cela est admissible pour réaliser un routeur logiciel, il ne peut en être question pour une implémentation matérielle. En effet la principale motivation qui conduit à la réalisation d'un circuit routeur est le gain en performances; et dans ce contexte l'emploi de tables de tailles indéterminées et dépendantes du réseau n'est pas désirable car cela empêche toute utilisation d'une mémoire intégrée au circuit et donc très rapide.

Parmi les méthodes de compactage de l'information de routage, le routage hiérarchique s'avère d'une construction très complexe. De plus les tables induites par cette technique sont encore dépendantes de la taille du réseau, et même si le gain est significatif, il reste cependant insuffisant ($O(mN^{1/m})$ où m est le nombre de niveaux hiérarchiques et N le nombre de noeuds). Il en va de même pour le routage par préfixes qui nécessite une information de routage de l'ordre du diamètre du réseau. Le lecteur peut se reporter à [Mug93] pour une synthèse de ces deux techniques. Il y trouvera également une étude plus détaillée du routage par intervalles, technique proposée par Santoro et Khatib [SanKha82] et reprise par van Leeuwen et Tan [LeTan87], que nous allons présenter brièvement et qui est une bonne solution au compactage de l'infor-

mation de routage.

Le routage par intervalles fonctionne à partir de tables inversées R'^{-1} telles qu'elles ont été introduites au début de ce chapitre; mais cette fois il s'agit de représenter les ensembles des destinations associés à chaque lien comme des intervalles. Comme les ensembles considérés ne sont pas tous composés de numéros de noeuds consécutifs, il faut trouver une méthode de renumérotation des noeuds, un schéma d'étiquetage, de telle sorte que tous les ensembles correspondent exactement à un intervalle.

Les intervalles manipulés dans ce contexte ne sont pas exactement linéaires; ils prennent en compte un ordre circulaire où le premier élément de l'ensemble des processeurs est le successeur du dernier. Soient p_1 et p_2 deux éléments de P (composé de N éléments numérotés de 0 à N-1), l'intervalle $[p_1,p_2)$ est défini de la manière suivante:

- si $p_1 < p_2$ alors $[p_1,p_2) = \{p_1, p_1+1, ..., p_2-1\}$;
- si $p1 > p2$ alors $[p_1,p_2) = \{0, 1, ..., p_2-1\} \cup \{p_1, ...,N-1\}$;
- si $p1 = p2$ alors $[p_1,p_2) = P$.

Le problème consiste donc à renommer les noeuds, de façon à ce que l'étiquette de tout lien corresponde à un intervalle de destinations. Cet étiquetage des liens, qui est donné par la fonction de routage, doit produire des intervalles deux à deux disjoints dont l'union couvre l'ensemble des étiquettes. A partir des étiquettes $e_0, e_1, ..., e_n$ associées aux liens d'un noeud, et classés de manière à avoir $e_0 < e_1 < ...< e_n$, les intervalles sont calculés comme suit:

- chaque lien étiqueté est associé à l'ensemble vide;
- au lien ayant reçu l'étiquette e_i, est associé l'intervalle $[e_i,e_{(i+1) \bmod n})$.

Comme exemple de mise en oeuvre, si nous appliquons un algorithme «e-cube» par dimensions décroissantes [SML93] dans une grille 3x3, nous obtenons l'étiquetage de la figure 3.8a. Selon le principe ci-dessus, les intervalles calculés sont donc ceux de la figure 3.8b. Dans cet exemple, le noeud 4 reçoit les étiquettes 0, 3, 5 et 6 et les intervalles suivants:

- pour le lien étiqueté 0 (connexion au noeud 1), l'intervalle [0,3) = {0, 1, 2};
- pour le lien étiqueté 3 (connexion au noeud 3), l'intervalle [3,5) = {3, 4};
- pour le lien étiqueté 5 (connexion au noeud 5), l'intervalle [5,6) = {6};
- pour le lien étiqueté 6 (connexion au noeud 7), l'intervalle [6,0) = {6, 7, 8}.

Cependant, trouver un schéma d'étiquetage adéquat est un problème qui n'admet malheureusement pas toujours de solution selon le réseau ou la fonction de routage employés. D'autre part du fait que cette méthode ne s'applique qu'à des tables qui ne considèrent que la destination comme élément de sélection dans le routage, le nombre de fonctions qu'elle est capable de représenter s'en trouve restreint. Pour pallier à cela on peut utiliser, par lien de sortie, autant d'intervalles que le noeud possède de liens d'entrée comme le préconise le **s**chéma d'**é**tiquetage **é**tendu (SEE) introduit dans [Mug93].

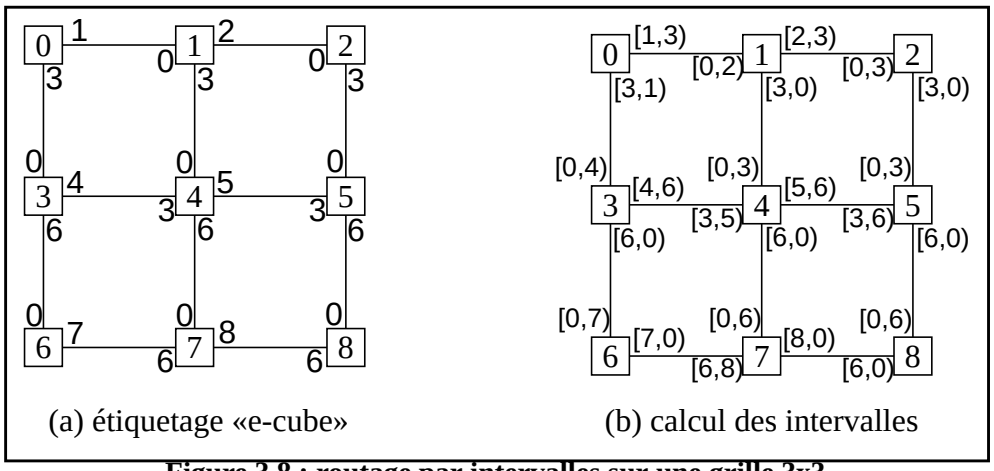


Figure 3.8 : routage par intervalles sur une grille 3x3.

3.5 Conclusion: le routeur de ParX

Lors de la conception du système PAROS [BFF89, BCD93], la définition d'un routeur fut l'un des principaux axes de recherche dans lequel notre équipe a travaillé depuis 1987. A cette époque bien peu de machines disposaient de routeurs directement intégrés au matériel, et pour celles qui en bénéficiaient il s'agissait de circuits propres à la topologie d'interconnexion des processeurs. Dans ce cadre nous avons introduit les deux fonctions de routage qui sont présentées dans ce chapitre, et qui répondent au critère important de correction pour tout réseau d'interconnexion. Celles-ci sont, l'une comme l'autre, utilisées dans le noyau de système ParX.

La technique d'acheminement choisie dans notre routeur est le «store-and-forward». Toutefois, à partir des études que notre équipe a menées sur le compactage de l'information de routage, et notamment celles de L. Mugwaneza sur le routage par intervalles, une intégration de ce routeur dans un circuit est possible. Nos résultats peuvent également être mis en oeuvre dans des routeurs matériels tels que les C104 par exemple [MTW93].

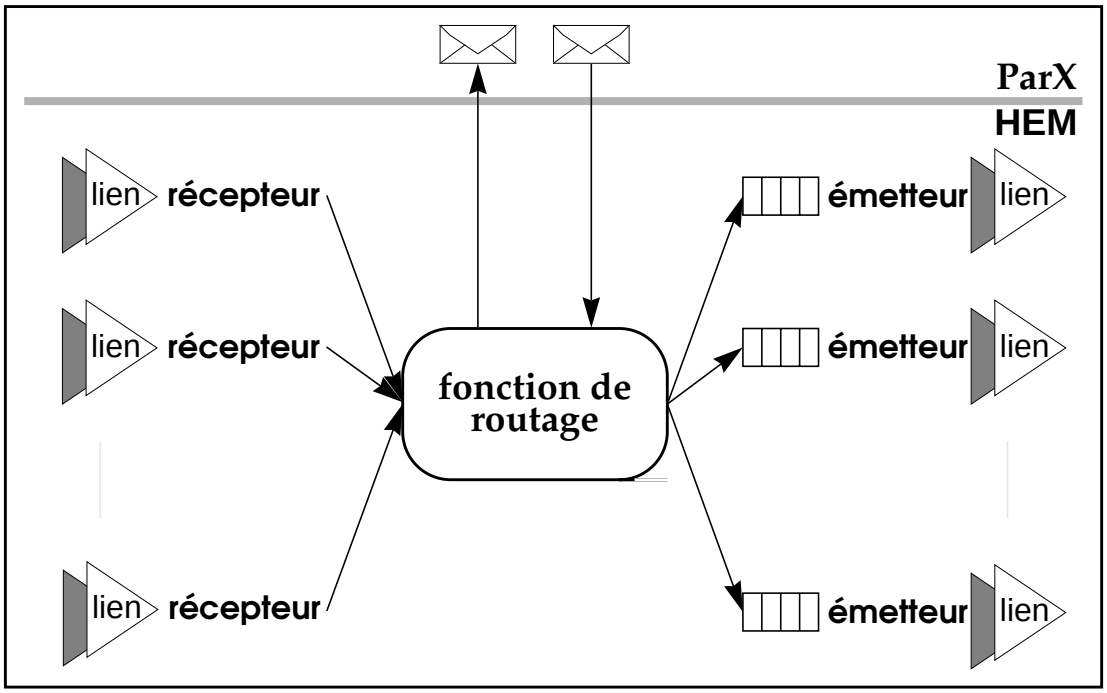


Figure 3.9 : structure du routeur de ParX.

Comme cela est représenté sur la figure 3.9, la structure de ce routeur, qui fait partie du noyau ParX au niveau des fonctions de plus bas niveau, se compose de trois éléments: les récepteurs et émetteurs de liens, et la fonction de routage. Alors qu'un récepteur dispose d'un tampon interne pour stocker les données reçues, tout émetteur est lié à un tampon externe, de taille fixe, dans lequel sont placés les messages, ou plus exactement les paquets, à émettre. La fonction de routage quant à elle se charge de rediriger les paquets sur les liens de sortie, ou vers la couche des protocoles du noyau. Elle est également responsable de l'arbitrage pour l'accès aux liens de sortie.

L'algorithme d'un récepteur de lien est assez simple et consiste à se mettre en attente de réception sur le lien; puis lorsque le paquet de données et son en-tête sont reçus de les transmettre à la fonction de routage:

```
PROC Récepteur( CHAN OF Structure.paquet Lien.Entrée,
                CHAN OF Structure.paquet Routage)
  Structure.paquet paquet:

  WHILE TRUE
    Lien.Entrée?paquet
    Routage!paquet
```

De même pour l'émetteur de lien, il suffit d'envoyer les messages qui sont soumis dans sa file d'attente (que l'on supposera accessible par des canaux):

```
PROC Emetteur(  CHAN OF Structure.paquet Fifo,
                CHAN OF Structure.paquet Lien.Sortie)
  Structure.paquet paquet:

  WHILE TRUE
    Fifo?paquet
    Lien.Sortie!paquet
```

Pour le coeur du routeur il s'agit tout d'abord d'effectuer une sélection équitable entre tous les récepteurs qui attendent de lui soumettre un paquet à router; puis de déterminer à l'aide de la table de routage, qui dans notre cas est directe et déterministe, le prochain lien à emprunter; et enfin de placer le paquet dans la file d'attente du lien obtenu. Dans notre algorithme nous avons supposé l'existence d'un pseudo-lien, implémenté par un canal et une file d'attente, pour assurer les communications entre le routeur et le noyau :

```
PROC FonctionRoutage(
          [DEGRE_NOEUD+1]CHAN OF Structure.paquet Entrées,
          [DEGRE_NOEUD+1]CHAN OF Structure.paquet Fifos)
  Structure.paquet paquet:
  INT lien.Suivant:
  INT i:
  WHILE TRUE
    ALT i = 0 FOR DEGRE_NOEUD + 1
      Entrées[i]?paquet
        SEQ
          lien.Suivant := Rn[i,Destination(paquet)]
          Fifos[lien.Suivant]!paquet
```

C’est dans cette structure que notre algorithme de diffusion doit s’intégrer. Mais avant de décrire notre algorithme et la manière dont il s’insère dans le routeur, nous commencerons par détailler les principales méthodes de routage pour la diffusion.

Chapitre 4 : Techniques de Routage pour la Diffusion

Ce chapitre présente les principales techniques de diffusion qui peuvent être mise en oeuvre dans les architectures massivement parallèles faiblement couplées. Nous avons dégagé ici six différentes méthodes qui regroupent un large spectre de techniques couramment utilisées dans les environnements distribués et parallèles. Ainsi nous aborderons successivement:

- la diffusion ***rayonnante***;
- la diffusion ***calculée***;
- la diffusion ***centralisée***;
- la diffusion sur un ***arbre***;
- la diffusion sur un ***anneau à jeton***;
- la diffusion par ***inondation***.

Dans notre présentation, outre le principe sur lequel est fondée chacune de ces méthodes, nous spécifierons leur algorithme général et nous nous pencherons sur leur efficacité. Pour évaluer cette efficacité nous utiliserons deux critères: l'**encombrement** induit sur le réseau de communication et le **temps** de diffusion. Un «bon» protocole sera un protocole qui offre un bon compromis pour ces deux critères; du meilleur il résultera un encombrement minimum ainsi qu'un temps optimal.

L'encombrement considéré sera déterminé par l'utilisation des ressources physiques, et plus précisément par la somme des liens de communication empruntés par un message pour atteindre toutes les destinations. En effet ce facteur est représentatif de la surcharge qui est introduite: plus il est faible, plus la probabilité de provoquer des congestions est faible. Comme la diffusion est par nature très consommatrice de ressources, il est important de limiter ce facteur afin de conserver une répartition équitable de la bande passante du réseau entre les diverses communications en concurrence.

En ce qui concerne le temps de diffusion le problème est plus complexe: il faudrait être capable d'évaluer les temps de latence en chaque noeud, temps induits par les communications en cours. Comme notre objectif n'est pas ici de donner une mesure exacte, qui prendrait en compte tous les conflits possibles ou le comportement de telle ou telle application, nous proposons de représenter ce temps comme une succession d'étapes. Le temps est alors mesuré par le nombre d'étapes nécessaires pour accomplir la totalité de la diffusion. En effet si nous occultons les retards qui peuvent être introduits, les transmissions de site en site d'un message diffusé se font étape par étape: dès lors que la source a expédié son message vers les destinations, se sont les noeuds adjacents qui retransmettent le message; puis la communication continue

ainsi de proche en proche. Dans cette vision nous pouvons considérer toute diffusion comme la construction d'un arbre d'enchaînements de communications. La racine de cet arbre est la source du message, les noeuds les sites atteints à chaque étape, et les branches les communications entre ces sites.

Pour illustrer la construction d'un tel arbre de diffusion, considérons un réseau de huit processeurs interconnectés selon le schéma de la figure 4.1a. Dans cette topologie, sans utiliser une méthode précise, nous pouvons diffuser un message du processeur 1 vers tous les autres par exemple de la manière suivante: tout d'abord le processeur expédie simultanément son message aux processeurs 2 et 3; puis il le transmet au processeur 4 alors que les processeurs 2 et 3 font de même respectivement pour les processeurs 5 et 6; enfin le processeur 4 communique le message au processeur 8 et les processeurs 5 et 6 le retransmettent en même temps au processeur 7. Le découpage de la diffusion est donc celui du tableau 4.1.

Etape	Communications	noeuds atteints
1	1 vers 2 1 vers 3	2, 3
2	1 vers 4 2 vers 5 3 vers 6	4, 5, 6
3	4 vers 8 5 vers 7 6 vers 7	8, 7

Tableau 4.1 : découpage en étapes d'une diffusion.

Nous obtenons alors l'arbre de la figure 4.1b, dont nous remarquerons qu'il peut contenir plusieurs occurrences d'un même processeur. Nous noterons également que le processeur 1 est son propre fils puisqu'il transmet dans cet exemple son message en deux étapes.

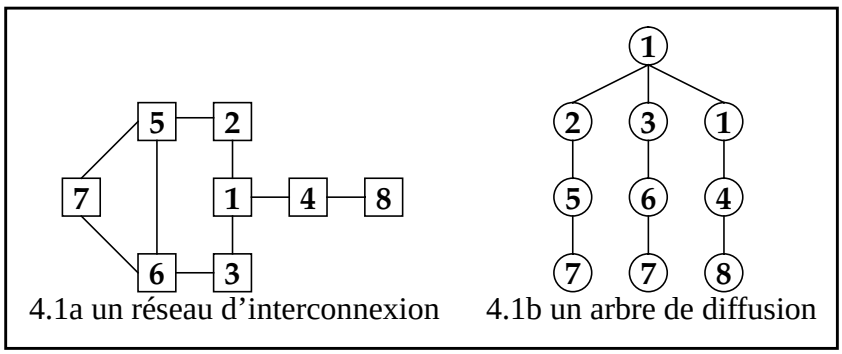


Figure 4.1 : un exemple de diffusion.

L'intérêt d'un tel arbre est de modéliser le déroulement des communications dans une diffusion, mais aussi et surtout d'évaluer le nombre d'étapes nécessaires pour atteindre toutes les destinations. Nous retiendrons donc la profondeur de l'arbre comme critère de temps dans les comparaisons des méthodes exposées.

4.1 Diffusion rayonnante

Le principe est ici assez simple puisqu'il consiste, pour la source, à envoyer un message à chaque destinataire. Avec cette méthode aucun routage spécifique à la diffusion n'est requis; c'est le routage point-à-point qui est exploité. Les algorithmes d'émission et de réception sont de ce fait tout aussi simples (avec $N>0$ le nombre total de sites du réseau):

```
PROC Diffusion.émission(
    Message m,
    [N] BOOL Destinataire,
    [N] CHAN OF Message Canaux.récepteurs)

  INT i:

  SEQ i = 0 FOR N
    IF
      Destinataire[i]
        Canaux.récepteurs[i]!m
      TRUE
        SKIP
:

PROC Diffusion.réception(
    CHAN OF Message Vers.application,
    CHAN OF Message Canal.réception)

  Canal.réception?m
  Vers.application!m
:
```

Il est évident que cette technique ne fait aucune optimisation tant pour l'encombrement du réseau que pour le temps de communication. Et même s'il est toujours possible de combiner les communications point-à-point de manière à opérer une factorisation des expéditions de messages, cette méthode reste limitée à un petit nombre de destinataires.

Pour une diffusion globale, vers tous les processeurs, dans une topologie d'interconnexion quelconque, l'évaluation de l'encombrement introduit est difficilement réalisable sans une connaissance approfondie de la fonction de routage point-à-point. Toutefois il est fort probable que ce paramètre soit nettement supérieur au nombre de liens du réseau: d'une part la quasi totalité des liens devraient être empruntés au moins une fois, certains même un grand nombre de fois dont notamment ceux connectés à la source.

De même, quantifier précisément la profondeur de l'arbre de diffusion dans le cas général n'est pas chose aisée. Cependant, sous l'hypothèse d'informations sur les distances qui séparent la source des autres noeuds, l'enchaînement des communications peut se faire des sites les plus éloignés aux sites les plus proches (les voisins directs) et ainsi obtenir un arbre de profondeur $(N-1)$, où N est la taille du réseau. En effet, comme les communications avec les sites les plus éloignés ne peuvent se faire sur une distance supérieure au nombre total de noeuds, cela correspond au nombre d'étapes nécessaires pour expédier tous les messages $(N-1)$.

Comme le montre les deux schémas des figures 4.2a et 4.2b, appliqué à une grille 4x4, avec un routage ***e-cube***, cet algorithme produit un encombrement de 48 liens pour une profondeur d'arbre de 15. La figure 4.2a représente la grille utilisée et la numérotation des processeurs dans celle-ci, ainsi que le nombre d'utilisation de chaque lien. L'arbre de la figure 4.2b est construit en envoyant le message à tous les destinataires dans l'ordre décroissant de leurs numéros.

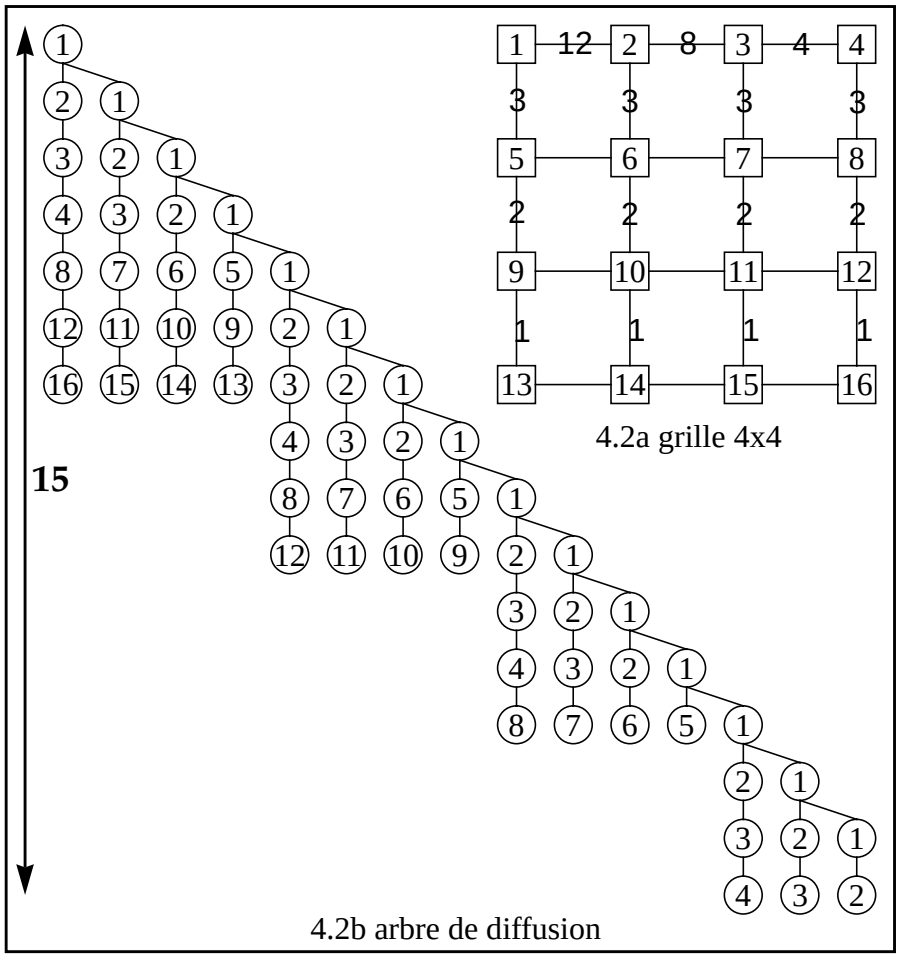


Figure 4.2 : diffusion rayonnante dans une grille 4x4.

En généralisant ces résultats à une grille carrée *n*x*n*, nous obtenons un encombrement égal à $n^2(n\text{-}1)$ liens et une profondeur d'arbre de $n^2\text{-}1$. En effet, l'encombrement se calcule en additionnant les utilisations des liens des colonnes:

$$n\sum_{i=1}^{n-1} i = \frac{n^2(n-1)}{2}$$

avec la somme des utilisations des liens de la première ligne:

$$n\sum_{i=1}^{n-1} i = \frac{n^2(n-1)}{2}$$

D'où un encombrement de $n^2(n\text{-}1)$.

Enfin nous noterons que d'une part la diffusion rayonnante est indépendante d'une fonction de routage particulière et peut donc être employée dans tous les cas, et que d'autre part il n'est pas nécessaire de l'intégrer au routeur, puisqu'elle peut être réalisée comme une simple primitive.

4.2 Diffusion calculée

Comme nous l'avons vu pour les communications point-à-point, sous l'hypothèse d'un réseau régulier, il est possible de trouver une fonction arithmétique de routage. Ces fonctions se caractérisent par une logique de numération des noeuds qui permet de calculer en chaque site le prochain noeud du chemin qu'un message doit emprunter. Il en résulte, pour tout noeud du réseau, un arbre recouvrant de communications composé de l'ensemble des chemins vers toutes les destinations. La propriété de ces arbres est qu'ils sont minimaux: de profondeur égale au diamètre du réseau, et qu'il n'y a qu'une et une seule occurrence de chaque noeud dans l'arbre.

Ce qui est valable pour les échanges point-à-point l'est, en principe, également pour la diffusion. Par voie de conséquence, les algorithmes de diffusion dans les réseaux réguliers ne font qu'exploiter les arbres donnés par leur fonction spécifique. Il existe de nombreux protocoles pour les principales topologies régulières utilisées; protocoles qui font l'objet d'une abondante littérature: [SaSc89a] pour l'anneau, [SaSc89a, TouPla90, Micha94] pour les grilles et les tores, [SaSc89a, SaSc89b, JoHo89, Toura89, StoWa90, DTW91] pour l'hypercube, ou encore [HOS92] pour les réseaux de De Bruijn.

La réalisation de ces protocoles passe par l'implantation d'un processus de routage des messages. De ce fait, pour la phase de routage il peut être nécessaire d'adjoindre au message des informations supplémentaires telles que, par exemple, l'identification de la source ou une orientation que le message doit suivre. Sur un anneau l'algorithme consiste à envoyer le message dans les deux sens de l'anneau, les noeuds étant équitablement répartis (avec N le nombre total de processeurs). Chaque noeud, après avoir transmis le message à l'application, le route vers le noeud suivant dans son sens de circulation. La diffusion se termine lorsque chacune des deux copies du message a traversé la moitié de l'anneau.

```
PROTOCOL Message_routé IS Message; INT: -- message + source
PROC Diffusion.anneau(
    CHAN OF Message De.Application, Vers.Application,
    CHAN OF Message_routé De.précédent, Vers.précédent,
                         De.suivant, Vers.suivant,
    INT n) -- identification du noeud local
  PAR
    WHILE TRUE -- source
      Message m:
      SEQ
        De.Application?m
        PAR
          Vers.précédent!(m,n)
          Vers.suivant!(m,n)
    WHILE TRUE -- sens positif
      Message m:
      INT source:
      SEQ
        De.précédent?(m,source)
        PAR
```

```
            Vers.Application!m
            IF
              (n <> (source + N/2) mod N)
                Vers.suivant!(m,source)
              TRUE
                SKIP
      WHILE TRUE -- sens négatif
        Message m:
        INT source:
        SEQ
          De.suivant?(m,source)
          PAR
            Vers.Application!m
            IF
              (n <> (source + N/2 + 1) mod N)
                Vers.précédent!(m,source)
              TRUE
                SKIP
:
```

Dans une grille *nxp* la méthode consiste à utiliser l'algorithme du routage en ***XY***:

```
PROC Diffusion.grille(
  CHAN OF Message De.Application, Vers.Application,
                  De.nord, Vers.nord, De.sud, Vers.sud,
                  De.est, Vers.est, De.ouest, Vers.ouest,
  INT x,y) -- coordonnée du noeud local

  PAR
    WHILE TRUE -- source
      Message m:
      SEQ
        De.Application?m
          PAR
            IF
              (x > 1)
                Vers.ouest!m
              TRUE
                SKIP
            IF
              (x < n)
                Vers.est!m
              TRUE
                SKIP
            IF
              (y > 1)
                Vers.nord!m
              TRUE
                SKIP
```

```
          IF
            (y < p)
              Vers.sud!m
            TRUE
              SKIP
IF
  (x > 1)
    WHILE TRUE -- routage en X, sens positif
      Message m:
      SEQ
        De.ouest?m
        PAR
          Vers.Application
          IF
            (x < n)
              Vers.est!m
            TRUE
              SKIP
  TRUE
    SKIP

IF
  (x < n)
    WHILE TRUE -- routage en X, sens négatif
      Message m:
      SEQ
        De.est?m
        PAR
          Vers.Application!m
          IF
            (x > 1)
              Vers.ouest!m
            TRUE
              SKIP
  TRUE
    SKIP

IF
  (y > 1)
    WHILE TRUE -- routage en Y, sens positif
      Message m:
      SEQ
        De.nord?m
        PAR
          Vers.Application!m
          IF
            (y < p)
              Vers.sud!m
            TRUE
              SKIP
```

```
                IF
                  (x > 1)
                    Vers.ouest!
                  TRUE
                    SKIP
                IF
                  (x < n)
                    Vers.est!m
                  TRUE
                    SKIP
        TRUE
          SKIP

      IF
        (y < p)
          WHILE TRUE -- routage en Y, sens négatif
            Message m:
            SEQ
              De.sud?m
              PAR
                Vers.Application!m
                IF
                  (y > 1)
                    De.nord!m
                  TRUE
                    SKIP
                IF
                  (x > 1)
                    Vers.ouest!m
                  TRUE
                    SKIP
                IF
                  (x < n)
                    Vers.est!m
                  TRUE
                    SKIP
        TRUE
          SKIP
:
```

Comme dernier exemple, nous donnons l'algorithme pour un hypercube de dimension *d*, où les processeurs sont numérotés de 0 à $N = 2^d$-1:

```
PROTOCOL Message_routé IS Message; INT:
PROC Diffusion.hypercube(
    CHAN OF Message De.Application, Vers.Application,
    [N+1] CHAN OF Message_routé De.proc, Vers.proc,
    INT n) -- identification du noeud local
  Message m:
  INT i, j, k, s:
```

```
  WHILE TRUE

    ALT
      Application?m -- source
        PAR i = 0 FOR d
          Vers.proc[n >< 2^i]!(m,source)
      ALT i = 0 FOR d -- routage
        De.proc[n >< 2^i]?(m,s)
          PAR
            Application!m
            SEQ
              k := (n >< i) / 2
              WHILE k > 0
                SEQ
                  Vers.proc[n >< k >< s]!(m,s)
                  k := k / 2
:
```

Par nature la diffusion calculée emploie un arbre minimal; il s'agit donc de protocoles optimaux pour les architectures régulières auxquelles elles sont destinées, tant pour l'encombrement qu'en ce qui concerne le temps. Avec une grille carrée *n*x*n*, nous obtenons un encombrement de n^2-1 liens et une profondeur d'arbre égale à 2*n*-2. Ce qui donne avec une grille 4x4, une profondeur d'arbre de 6 et un encombrement de 15 liens, comme nous pouvons le voir dans la figure 4.3.

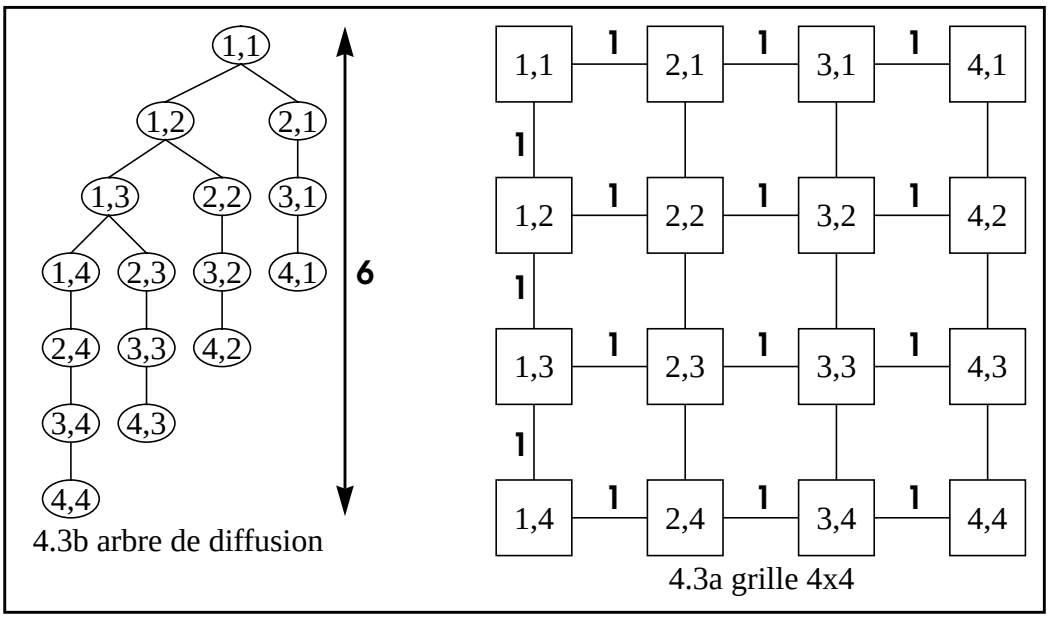


Figure 4.3 : diffusion calculée dans une grille 4x4.

L'attrait de cette technique est qu'elle détermine pour toute source un arbre de diffusion optimal, et ce par simple calcul. Par contre ce calcul n'est valable qu'avec des topologies régulières; et non seulement il n'existe pas de méthode générale, mais de plus la plupart des réseaux ne possèdent pas une telle propriété. Pour conserver un tel fonctionnement, il est cependant possible de précalculer un arbre de diffusion pour chacune des sources; mais alors l'utilisation de tels arbres peut engendrer des interblocages, soit avec le routage point-à-point,

soit entre des diffusions concurrentes. Par ailleurs il se pose le problème du stockage de ces arbres qui sont nécessairement dépendants de la taille du réseau.

4.3 Diffusion centralisée

Pour contourner les problèmes posés par le calcul d'arbre selon le principe précédent, une solution consiste à ne conserver qu'un seul arbre calculé pour une unique source. Le choix de cette source se porte généralement sur le centre du graphe d'interconnexion; l'arbre considéré est alors l'arbre minimal depuis cette source. La résolution des interblocages avec la fonction de routage point-à-point est aisée: il suffit d'introduire une classe de tampons supplémentaire affectée à la diffusion.

La diffusion d'un message depuis n'importe quel site se fait alors en deux phases: l'acheminement du message vers la racine de l'arbre, puis la diffusion de celui-ci. Si nous occultons le détail de l'acheminement point-à-point, la modélisation d'un tel protocole fait appel à deux éléments: une fonction de communication point-à-point vers la racine de l'arbre pour l'application, et un processus de routage dans l'arbre. Dans l'algorithme ci-après, nous avons distribué l'arbre de manière telle que chacun des *N* noeuds reçoive dans le tableau **`Sous.arbre`** les canaux de communication vers ses *d* fils dans l'arbre.

```
PROC Diffusion.centralisée(
    CHAN OF Message De.Application, Vers.racine)
  Message m:
  WHILE TRUE
    SEQ
      De.Application?m
      Vers.racine!m
:
PROC Racine.diffusion([N] CHAN OF Message De.sources,
    [d] CHAN OF Message Sous.arbre,
    CHAN OF Message Vers.Application)
  Message m:
  WHILE TRUE
    ALT i = 0 FOR N
      De.sources[i]?m
        PAR
          Vers.Application!m
          PAR j = 0 FOR d
            Sous.arbre[j]!m
:
PROC Noeud.diffusion(
    CHAN OF Message Vers.Application, De.père,
    [d] CHAN OF Message Sous.arbre)
  Message m:
  WHILE TRUE
    SEQ
      De.père?m
      PAR i = 0 FOR d
        Sous.arbre[i]!m
:
```

L'arbre n'est certes pas l'unique structure de diffusion utilisable avec cette méthode, mais dans les architectures parallèles à échange de messages il se montre le mieux approprié pour la diffusion centralisée. M. Raynal, dans [Raynal91], donne un exemple de protocoles qui suit ce principe. L'analyse de ce protocole, quoique très simple, montre deux faiblesses non négligeables qui sont la *panne du site central* et le *goulot d'étranglement* que celui-ci représente. Afin de pallier à ces défauts, l'auteur propose d'y adjoindre un mécanisme d'élection de la racine.

Il est également possible de s'inspirer de la solution proposée par Chang et Maxemchuk pour une diffusion assurant un ordre total [ChaMa84]. Selon cette méthode deux phases s'alternent: une *phase normale* et une *phase de reformation*. Durant la phase normale, lorsqu'un message est diffusé il est transmis aux applications; puis le rôle du noeud central est transféré à un autre parmi une liste de noeuds possibles. La phase de reformation n'intervient que lors de pannes ou d'erreurs, et consiste à redéfinir la liste des sites centraux aptes à diffuser les messages, et éventuellement à reconstruire l'arbre. [KTHB89], sous des hypothèses de pannes et d'erreurs moins fortes, ont développé un protocole similaire et plus efficace. Il se base sur un mécanisme d'historique des messages: tout message émis est conservé dans l'historique jusqu'à son acquittement (différé) par tous les sites; la détection de la perte d'un message provoque sa réémission depuis l'historique; enfin, lorsque cet historique est plein tous les noeuds entrent dans une phase spéciale qui permet de le vider.

En terme d'efficacité, une telle méthode produit une profondeur d'arbre de diffusion réelle proportionnelle au diamètre du réseau: la moitié du diamètre pour la communication avec le site central (centre du graphe), et de nouveau la moitié du diamètre pour la diffusion dans l'arbre. De même nous pouvons évaluer l'encombrement à la somme de la moitié du diamètre (communication avec la racine) et du nombre de noeuds (diffusion dans l'arbre). Mais cette évaluation ne tient pas compte des surcoûts introduits par les deux protocoles cités un peu plus haut.

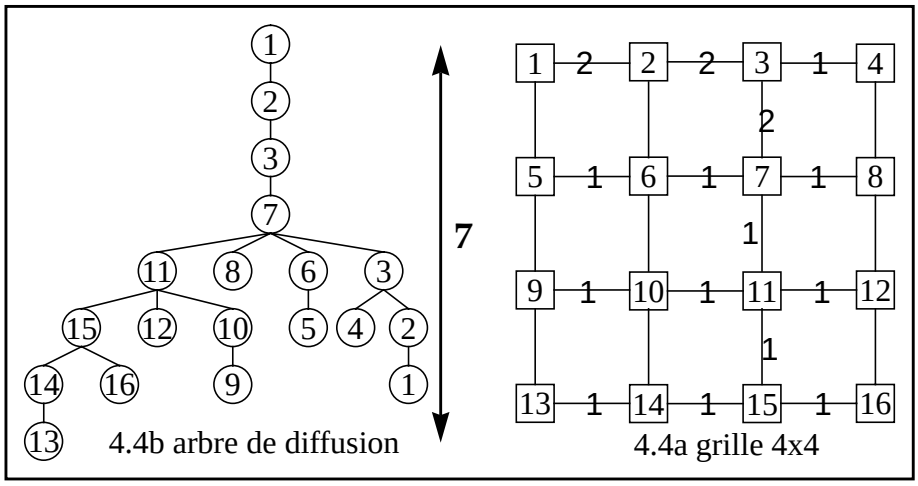


Figure 4.4 : diffusion centralisée dans une grille 4x4.

Nous retrouvons ces résultats pour la grille carrée *n*x*n*, avec une profondeur d'arbre de 2*n*-2 si *n* est impair ou 2*n*-1 si *n* est pair, et un encombrement de (n-1)+(n^2-1)=n^2+n-2 liens. C'est-à-dire avec une grille 4x4, un encombrement de 18 liens et une profondeur d'arbre égale à 7 (cf. figure 4.4).

4.4 Diffusion dans un arbre

Pour la méthode précédente nous avons utilisé un arbre dans lequel la diffusion était opérée uniquement à partir de la racine. Il apparait clairement que la première communication vers la racine introduit une redondance de noeuds dans l'arbre final de diffusion; d'une part ce transfert n'est pas exploité pour la communication globale, et d'autre part cela augmente sensiblement l'encombrement total. Pour pallier ce problème, et plutôt que d'éliminer ces noeuds dans la diffusion depuis la racine - ce qui n'est pas toujours possible -, l'arbre peut être utilisé comme structure de diffusion pour tous les sites.

Le protocole de diffusion est donc transformé en un parcours d'arbre, où chaque noeud dispose de deux canaux de communication pour chacun de ses *d* fils, et deux autres vers son père. Pour simplifier nous donnons une expression de l'algorithme dans laquelle *d* peut être nul et les canaux de communication avec le père de la racine sont spéciaux: le canal en émission se comporte comme un «absorbant» de données, et celui en réception comme une entrée/sortie indéfiniment bloquante.

```
PROC Diffusion.arbre(
  CHAN OF Message De.Application, Vers.Application,
                  De.père, Vers.père,
  [d] CHAN OF Message De.fils, Vers.fils)

  Message m:

  WHILE TRUE
    ALT
      De.Application?m
        PAR
          Vers.père!m
          PAR i = 0 FOR d
            Vers.fils[i]!m

      De.père?m
          PAR
            Vers.Application!m
            PAR i = 0 FOR d
              Vers.fils[i]!m

      ALT i = 0 FOR f
        De.fils[i]?m
          PAR
            Vers.Application!m
            Vers.père!m
            PAR j = 0 FOR d
              IF
                j <> i
                  Vers.fils[j]!m
                TRUE
                  SKIP
:
```

Même si la racine de l'arbre n'est plus en charge d'un traitement des requêtes de diffusion, puisque chaque source se trouve être ici une nouvelle racine, la racine de l'arbre reste cependant un point de forte congestion. Nous retrouvons ici le phénomène qui se produit avec le routage point-à-point, mais auquel s'ajoute le trafic élevé provoqué par la diffusion.

En terme de performances la profondeur de l'arbre réel de diffusion varie entre la moitié du diamètre pour la racine, au diamètre du réseau pour les feuilles, en considérant un arbre optimal calculé à partir du centre du graphe. L'encombrement est par contre sensiblement réduit par rapport à la méthode centralisée puisqu'il n'est plus égal qu'au nombre de noeuds.

Pour la grille carrée *n*x*n*, nous obtenons une profondeur d'arbre comprise entre *n*-1 et 2*n*-1 pour un encombrement de n^2-1 liens. Ce qui donne sur la grille 4x4, pour une diffusion du processeur 1 dans l'arbre de racine le processeur 7, une profondeur d'arbre égale à 7 et un encombrement de 15 liens (cf. figure 4.5).

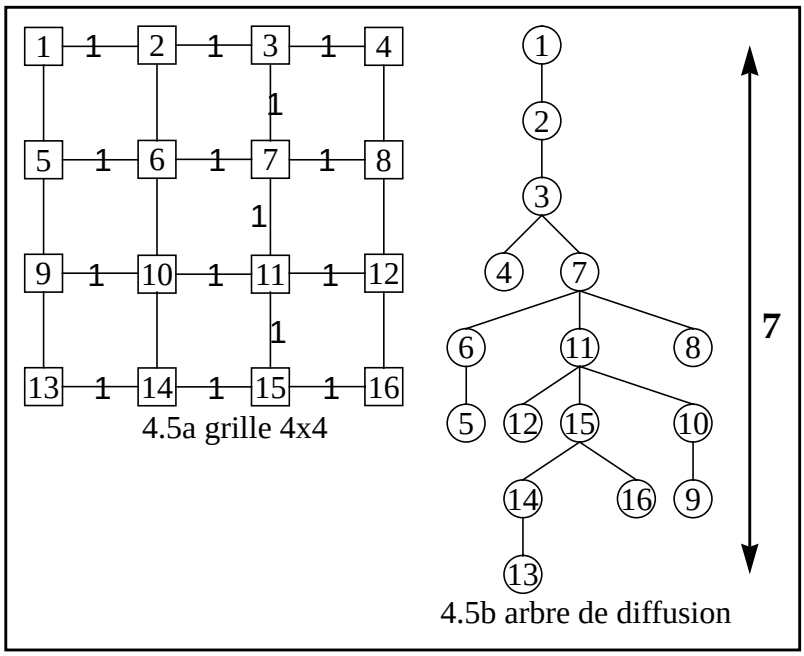


Figure 4.5 : diffusion dans un arbre pour une grille 4x4.

4.5 Diffusion sur un anneau à jeton

Dans cette méthode le réseau d'interconnexion est vu comme un anneau unidirectionnel, soit virtuellement, soit par extraction d'un cycle hamiltonien lorsque le graphe en admet un (cf. chapitre précédent, §3.3). Cet anneau est utilisé pour faire circuler un jeton de noeud en noeud, jeton qui contient les messages à diffuser par les différents sites (au plus un message par site). A chaque passage du jeton le noeud visité extrait les différents messages, retire son précédent message s'il existe, et enfin, le cas échéant, y dépose son nouveau message.

Le jeton peut être structuré comme un tableau de messages, de taille *N* (nombre de processeurs). Le protocole se résume alors à un processus de manipulation du jeton, qui communique avec l'application, et les deux sites respectivement qui le précède et qui le suit dans le sens de l'anneau.

```
PROTOCOL Jeton IS [N] Message:

PROC Diffusion.anneau_à_jeton(
    CHAN OF Message De.Application, Vers.Application,
    CHAN OF Jeton De.précédent, Vers.suivant,
    INT n) -- identification du noeud local

  Message m:
  Jeton j:

  SEQ
    m := NIL -- message nul
    WHILE TRUE
      ALT

        De.précédent?j
          SEQ
            SEQ i = 0 FOR N
              IF
                i = n
                  SEQ
                    j[i] := m
                    m := NIL
                j[i] <> NIL
                  Vers.Application!j[i]
                TRUE
                  SKIP
            Vers.suivant!j

        (m = NIL) & De.Application?m
          SKIP
:
```

Tel qu'il est décrit ci-dessus, le protocole de diffusion ne tient pas compte d'éventuelles pannes, erreurs ou pertes du jeton. Le lecteur trouvera dans [Raynal91] deux solutions à ces problèmes: la régénération du jeton sur détection de perte, et la circulation de deux jetons.

L'évaluation de cet algorithme est simple pour les réseaux qui admettent un cycle hamiltonien; dans ce cas il produit une profondeur d'arbre de diffusion égale à l'encombrement, soit au nombre de noeuds. Pour toutes les topologies sans cycle hamiltonien, ces paramètres sont soumis à la fonction de routage sur laquelle la virtualisation de l'anneau est effectuée. Les valeurs précédentes sont donc dans cette situation des approximations très minimalistes; il faudrait en plus tenir compte de la longueur de tous les chemins entre les noeuds adjacents dans l'anneau virtuel.

Appliqué à la grille carrée *n*x*n*, nous obtenons une profondeur d'arbre de n^2 pour un encombrement égal à n^2 liens. Ce qui donne avec la grille 4x4 une profondeur d'arbre et un encombrement égaux à 16 (voir figure 4.6).

Il faut cependant relativiser ces résultats et valeurs, car le jeton est capable de véhiculer simultanément autant de diffusions qu'il y a de processeurs, au prix certes d'une taille de jeton bien

plus importante que celle d'un message. La taille du jeton est justement le problème majeur de cette technique puisqu'elle est directement dépendante du nombre de noeuds du réseau, et de ce fait elle rend cette méthode plus difficile à utiliser dans un environnement massivement parallèle où le nombre de processeurs peut varier dynamiquement.

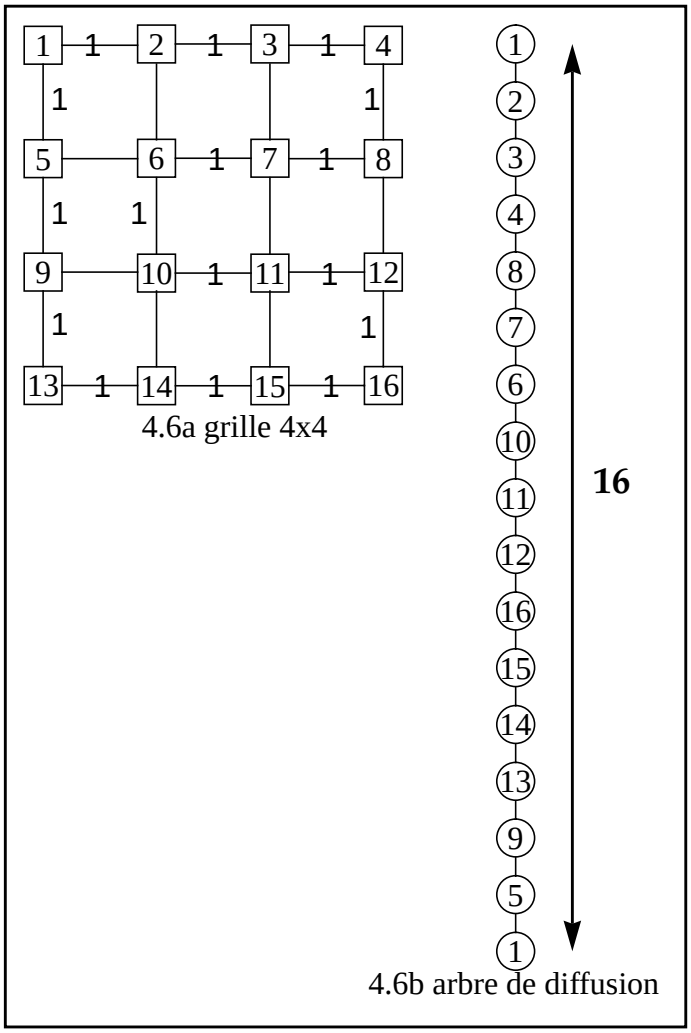


Figure 4.6 : diffusion sur un anneau à jeton pour une grille 4x4.

4.6 Diffusion par inondation

La technique consiste ici à emprunter tous les liens du réseau au moins une fois; cette propagation nous assure ainsi que tous les noeuds recevront le message. Pour cela la transmission du message se fait de proche en proche en appliquant la règle suivante: lorsqu'un site reçoit le message sur l'un de ses liens, il le retransmet sur tous ses autres liens. Le processus est initié par la source qui envoie son message sur tous ses liens. Pour s'assurer que chaque site n'effectue la retransmission qu'une et une seule fois, toute nouvelle réception du même message doit être ignorée. Par voie de conséquence le protocole se termine lorsque chaque noeud à reçu le message de tous ses voisins.

Dans notre algorithme nous avons supposé l'existence de structures de données ensemblistes avec les opérateurs traditionnels: union, appartenance, etc. De plus, pour communiquer chaque noeud du réseau dispose de deux canaux de communication avec chacun de ses *d* voisins. Nous avons alors l'algorithme ci-après.

```
PROC Diffusion.inondation(
    CHAN OF Message De.Application, Vers.Application,
    [d] CHAN OF Message De.voisin, Vers.voisin)

  SET OF Message reçus:
  Message m:

  SEQ
    reçus := ∅
    WHILE TRUE
      ALT

        De.Application?m
          SEQ
            reçus := reçus ∪ {m}
            PAR i = 0 FOR d
              Vers.voisin[i]!m

        ALT i = 0 FOR d
          De.voisin?m
            IF
              m ∉ reçus
                SEQ
                  reçus := reçus ∪ {m}
                  PAR
                    Vers.Application!m
                    PAR j = 0 FOR d
                      IF
                        j <> i
                          Vers.voisin[j]!m
                        TRUE
                          SKIP
              TRUE
                SKIP
:
```

Les études menées sur cette technique sont essentiellement dues à Y. K. Dalal et R. M. Metcalfe [DaM78] et D. M. Topkis [Top85]. Dalal et Metcalfe proposent une amélioration de l'algorithme qui consiste, pour un site intermédiaire, à n'effectuer la rediffusion d'un message que lorsque celui-ci arrive par le lien qui appartient au plus court chemin entre le noeud courant et la source; et ainsi le stockage des messages reçus devient inutile. Ils suggèrent également de calculer un arbre de diffusion qui est celui des plus courts chemins de toutes les destinations vers la source de diffusion. En conséquence de quoi chaque noeud disposera, selon la source, d'une association entre le lien d'entrée et l'ensemble des liens de sortie à emprunter. De cette manière la forte duplication des messages induite par la méthode est évitée.

Topkis quant à lui généralise l'algorithme à des communications **tous-vers-tous**. Les duplications sont évitées dans ce cas par alternance d'une phase de *transmission* et d'une phase de *signalisation*. Durant cette dernière chaque processeur informe ses voisins des messages qu'il possède déjà afin qu'ils ne les lui retransmettent de nouveau pendant la phase de transmission.

Quoiqu'il en soit, appliquée telle que, cette méthode produit un encombrement proportionnel à la somme des degrés d_i de chacun des n noeuds du réseau:

$$1 + \sum_{i=1}^{n} (d_i - 1)$$

L'encombrement est donc proportionnel au nombre de liens (l): 1+l-n. Ce qui donne, lorsque tous les noeuds ont le même degré d: n(d-1)+1. La profondeur de l'arbre de diffusion est au moins égale au diamètre du graphe d'interconnexion, auquel s'ajoute un nombre variable d'étapes nécessaires à la terminaison de l'algorithme.

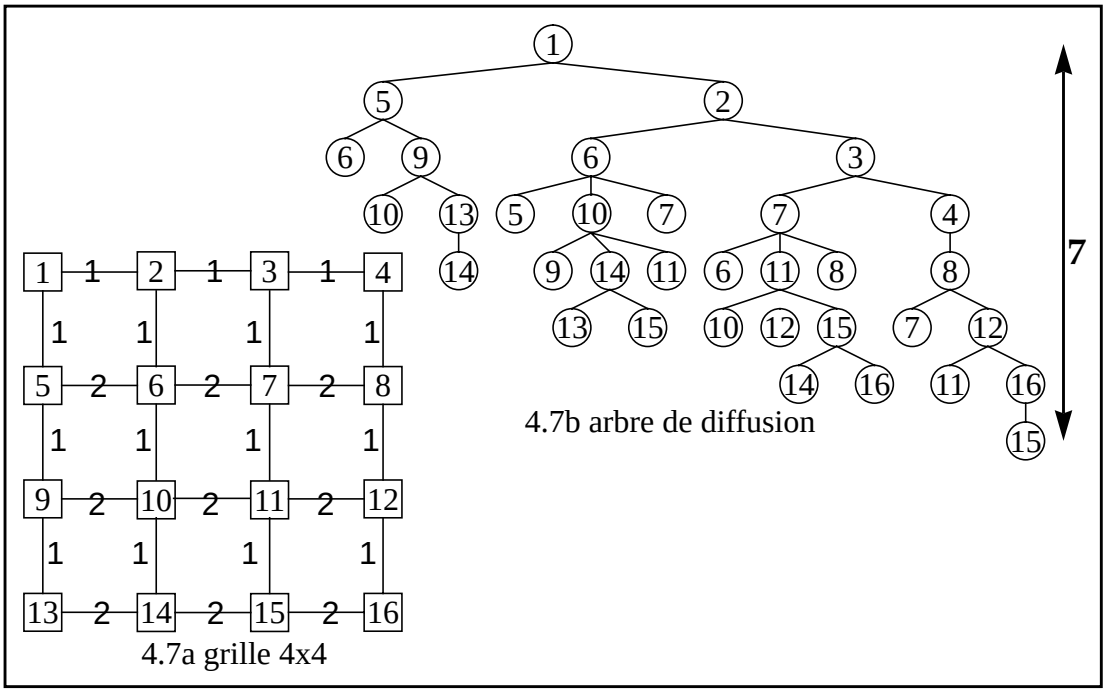


Figure 4.7 : diffusion par inondation dans une grille 4x4.

Pour une grille 4x4, comme le montre la figure 4.7, la profondeur de l'arbre est égale à 7, et l'encombrement est de 33 liens. Ces résultats se généralisent aisément à une grille carrée nxn, avec une profondeur d'arbre de 2n-1 et un encombrement égal à la somme des liens de toutes les colonnes, ceux de la première ligne et le double de ceux des lignes restantes, soit:

$$n\,(n-1) + (n-1) + 2\,(n-1)\,(n-1) = 3n^2 - 4n + 1$$

De cette technique nous retiendrons son indépendance vis-à-vis de la taille du réseau dans le routage des messages, mais qu'elle ne traite absolument pas le problème d'interblocages, et qu'elle produit un nombre très important de copies inutiles du même message.

4.7 Conclusion

Nous venons d'étudier dans ce chapitre les principales méthodes de diffusion existantes. Il ressort de cette étude que de nombreux protocoles ont été développés avec des motivations initiales diverses: certains réalisent une diffusion fiable lorsque l'acheminement ne l'est pas, atomique, qui assurent un ordre de réception donné, ou encore dont les sources peuvent être multiples; tandis que d'autres s'attachent à n'effectuer qu'un acheminement particulier. Pour notre part nous pensons que l'on ne peut, dans le cadre du parallélisme massif, construire un protocole sans le fonder sur un acheminement «efficace» et **correct**, tant en ce qui concerne l'encombrement du réseau qu'en temps de communication. Un autre facteur essentiel est l'indépendance de la fonction d'acheminement et de routage vis-à-vis du réseau, et principalement du nombre de noeuds.

Méthode	Cas général		Grille nxn	
	encombrement	profondeur	encombrement	profondeur
Rayonnante	$\geq Nd / 2$	$\approx N - 1$	$n^2 (n - 1)$	$n^2 - 1$
Calculée			$n^2 - 1$	$2n - 2$
Centralisée	$\approx D/2+N$	$\approx D$	$n^2 + n - 2$	$2n - 1$
Arbre	N	D	$n^2 - 1$	$2n - 1$
Anneau à jeton	$\geq N$	$\geq N$	n^2	n^2
Inondation	$N(d-1)+1$	$\geq D$	$3n^2-4n+1$	$2n - 1$

Tableau 4.2 : comparaison des principales méthodes de diffusion.

Face aux contraintes de performances que nous nous sommes fixées, nous pouvons maintenant comparer les différentes techniques exposées tout au long de ce chapitre. Le tableau 4.2 présente cette comparaison pour des réseaux quelconques et pour les grilles bidimensionnelles carrées. Dans le cas général il s'agit d'un réseau de N processeurs de degré d (le nombre total de liens est donc $Nd/2$), et de diamètre D. Nous ne donnons ici qu'une approximation des paramètres d'encombrement et de profondeur d'arbre de diffusion. Nos comparaisons se portent donc également sur un réseau particulier et très utilisé: la grille carrée nxn.

Le tableau 4.2, montre clairement l'inadaptation de la méthode rayonnante au contexte du parallélisme massif. Inversement les techniques de diffusion calculées s'avèrent les plus efficaces; mais elles ne sont pas applicables à des topologies quelconques. La solution de l'arbre ou celle centralisée se montrent efficaces par rapport aux paramètres choisis, mais il faut cependant tenir compte de la congestion et du goulot d'étranglement qu'elles introduisent. L'anneau à jeton apparait alors être un bon compromis, car même si les résultats obtenus sont moyens, ils doivent être relativisés puisqu'ils correspondent à plusieurs communications simultanées. Enfin l'inondation donne de bonnes performances pour ce qui est du temps mais produit par contre un encombrement très important; de plus, si elle ne peut être utilisée telle quelle sans risque d'interblocages, elle est totalement indépendante de la topologie d'interconnexion.

Chapitre 5 : Routage Correct pour la Diffusion

Dans ce chapitre nous présenterons un algorithme de diffusion, ou plus exactement une famille d'algorithmes fondés sur le même principe. Le protocole que nous avons construit conserve des propriétés similaires à celles du routage point-à-point:

- *validité* et *atomicité*: tous les messages parviennent à leurs destinations, et tous les noeuds reçoivent tous les messages qui leur sont destinés;
- *terminaison*: un message ne circule pas continuellement dans le réseau, les duplications sont donc éliminées;
- *absence d'interblocage*: la diffusion d'un message n'entraîne pas d'interblocages vis-à-vis d'autres communications point-à-point ou diffusions concurrentes;
- *absence de famine* et *équité*: l'algorithme conserve les propriétés du routage point-à-point.

L'ensemble des solutions que nous avons présentées au chapitre précédent respectent toutes ces propriétés, à l'exception toutefois de l'inondation qui ne peut garantir l'absence d'interblocage - seule la variante proposée par [DaM78] qui consiste à calculer un arbre de diffusion par source peut assurer cette propriété. Le choix de l'une ou l'autre de ces techniques comme méthode de base pour un algorithme de diffusion fut donc guidé par son adaptation aux différentes architectures massivement parallèles.

Pour cela nous avons écarté la diffusion rayonnante qui est non seulement insatisfaisante sur le plan des performances, et qui de surcroît se montre particulièrement inappropriée au parallélisme massif. Inversement les techniques de diffusion calculée apparaissent être les plus efficaces; mais elles ne sont cependant valables que pour les réseaux réguliers pour lesquels elles ont été conçues et ne peuvent donc pas être utilisées pour toutes les architectures - notamment celles qui sont reconfigurables où la topologie d'interconnexion peut varier dynamiquement et ne pas être régulière. En conséquence nous ne retiendrons pas cette méthode pour notre protocole; nous verrons toutefois que notre famille d'algorithmes englobe ces techniques.

Comme nous avons pu le remarquer, utiliser un arbre ou prendre un site central (avec un arbre associé) pour effectuer la diffusion produit des résultats assez comparables. L'une et l'autre de ces techniques se basent sur le calcul d'un arbre et sont en cela parfaitement adaptables à toute architecture parallèle. Le frein à leur utilisation ne vient pas de la structure du réseau d'interconnexion, mais du nombre de noeuds qu'il comprend: plus ce nombre est grand, plus leurs performances se dégraderont. En effet, le nombre de requêtes que la racine doit traiter ou router croît avec le nombre de noeuds, et la probabilité que ce nombre de requêtes produise un goulot d'étranglement devient d'autant plus élevée qu'il y a de processeurs. Le système d'échange de messages étant l'un des facteurs les plus cruciaux des systèmes parallèles, nous

ne pouvons utiliser l'une ou l'autre pour construire notre algorithme.

Ce problème de saturation ne se retrouve pas avec l'anneau à jeton, qui intrinsèquement permet de réguler le trafic généré par les diffusions. Par contre, à l'exception de la méthode rayonnante il s'agit de celle qui présente les plus faibles performances du point de vue du temps de diffusion. Même si les résultats obtenus pour ce critère peuvent être relativisés par le nombre de communications simultanées que peut véhiculer un jeton, une diffusion s'effectuera toujours en un nombre d'étapes important. De plus il faut également prendre en compte qu'un réseau peut ne pas admettre de cycle hamiltonien, ce qui rend l'algorithme bien moins efficace encore.

La dernière méthode que nous pouvons utiliser est celle de l'inondation dont nous avons vu qu'en plus de produire un trafic très élevé, elle ne respecte pas la propriété d'absence d'interblocage. Par contre elle présente l'avantage d'être totalement indépendante de la structure du réseau et du nombre de noeuds de celui-ci, ce qui la rend très attractive pour les architectures massivement parallèles. Par ailleurs, contrairement aux autres méthodes plus rigides, cette technique peut aisément être modifiée ou adaptée afin, entre autres, de réduire le nombre de duplications inutiles des messages.

Notre choix s'est donc orienté sur l'inondation pour construire un algorithme de routage correct pour la diffusion de messages. A partir de ce choix, notre objectif principal fut d'aboutir à un protocole d'inondation qui n'introduise aucun interblocage; puis nous avons cherché à déterminer comment minimiser l'encombrement induit par la diffusion d'un message. Nous terminerons ce chapitre par une étude sur l'utilisation du routage par intervalles dans notre algorithme, et sur sa capacité à être intégré à un routeur matériel.

5.1 Représentation de la fonction de diffusion

La possibilité de provoquer des interblocages est donc l'obstacle principal à l'utilisation de l'inondation. Nous retrouvons là une problématique similaire à celle du routage point-à-point que nous avons présenté au chapitre 3. Nous avons vu qu'il est nécessaire de définir une fonction de routage pour laquelle le graphe de dépendances soit acyclique. Notre idée est d'appliquer le même raisonnement avec l'inondation, et ainsi de définir une fonction de diffusion qui rend l'acheminement correct. Nous pouvons d'ailleurs remarquer que l'inondation est la seule des techniques de diffusion pour laquelle cette démarche est possible; les autres intègrent en effet une fonction figée par la méthode.

Pour garantir l'absence d'interblocage au sein d'une même diffusion ou entre diffusions concurrentes une fonction propre à la diffusion, indépendante du routage point-à-point est une solution. Par contre pour traiter les interblocages vis-à-vis des autres communications point-à-point, la fonction de diffusion doit tenir compte des interdictions imposées par le routage point-à-point. Nous avons donc définit notre diffusion par rapport à la fonction de routage point-à-point.

Ainsi, si f est une fonction de routage point-à-point sans interblocage (cf. chapitre 3, section 3.2.1), la fonction de diffusion F associée à f est définie comme suit:

$$F : L \times P \times L \rightarrow B$$

$$(e, n, s) \mapsto \begin{cases} vrai \; si \; \exists d \in P, f(e, n, d, s) = vrai \\ faux \; sinon \end{cases}$$

où P est l'ensemble des noeuds, L l'ensemble des liens de communication et B l'ensemble des booléens, et où F autorise au noeud n la rediffusion d'un message, arrivé par le lien e, sur tous les liens s pour lesquels il existe au moins un chemin vers la destination d autorisé par f qui passe par e puis par s.

Comme avec la fonction de routage point-à-point, F peut être distribuée à travers le réseau de telle sorte que tout noeud n reçoive une restriction F_n de F, où:

$$F_n : L \times L \rightarrow B$$
$$(e, s) \mapsto F(e, n, s)$$

Nous pouvons alors encoder en chaque noeud une table de diffusion sous forme d'un tableau D_n à une seule dimension où pour chaque lien d'entrée e:

$$D_n[e] = \{ s \in L, F_n(e, s) = vrai \}$$

5.2 Diffusion correcte

A partir de la fonction de diffusion F et des tables de diffusion D_n associées telles que nous venons de les définir, nous sommes capables de construire un protocole d'inondation correct. Pour cela le routeur à diffusion se compose d'un processus (lui-même parallèle) qui effectue deux types d'actions:

- l'**injection** d'un message en provenance de l'application. Le processeur, qui est alors la source du message, le transmet sur tous ces liens;
- le **routage** des messages. Lorsqu'un message est reçu sur l'un des liens e du noeud n, il est transmis à l'application, et rediffusé sur tous les liens s autorisés par la fonction de diffusion, c'est-à-dire $D_n[e]$.

Notons que D_n peut être aisément calculée à partir des tables de routage point-à-point:

$$\forall e \in L, D_n[e] = \{ s, \exists d \in P, s \in R_n[e, d] \}$$

ou, s'il on dispose de tables inversées:

$$\forall e \in L, D_n[e] = \{ s, R_n^{-1}[e, s] \neq \emptyset \}$$

ou encore, lorsque les tables font abstraction du lien d'arrivé:

$$\forall e \in L, D_n[e] = \{ s, R'^{-1}_n[s] \neq \emptyset \}$$

5.2.1 L'algorithme de diffusion correcte

L'algorithme ainsi ébauché, il nous faut maintenant éviter que l'application ne reçoive en chaque site plusieurs exemplaires du même message, et réduire le plus possible des duplications inutiles [DeMu93a]. Nous devons donc doter notre algorithme d'un mécanisme d'élimination des messages. Nous avons choisi pour cela de numéroter séquentiellement les messages à la source et d'estampiller chaque message par ce numéro d'ordre: *id*, et par l'identification de la source : *o*; ce qui associe un identificateur unique au message.

Pour ne pas transmettre aux destinataires plus d'une fois chaque message, le routeur gère une table *iddel* des derniers messages délivrés en fonction de la source: *iddel*[*o*] représente le numéro d'ordre du dernier message reçu de la source *s*. La condition de livraison d'un message estampillé par (*id*,*o*) est alors que son numéro soit strictement supérieur à l'entrée correspondante dans la table: *id* > *iddel*[*o*].

L'élimination de la plus grande part des duplications inutiles repose sur le même principe: chaque noeud dispose d'une table *idsuiv* qui indique les numéros des prochains messages devant transiter sur chacun des liens et en provenance des différentes sources: *idsuiv*[*l*,*o*] identifie le prochain message de la source *o* qui pourra emprunter le lien *l*. L'ensemble des liens de rediffusion d'un message estampillé par (*id*,*o*) et arrivé au noeud *n* par le lien *e* est donc:

$$\{ l \in D_n[e], id \geq idsuiv[e, o] \}$$

Certes cette règle n'est pas suffisante pour assurer une élimination totale des copies superflues; mais elle ne peut l'être si nous voulons garantir la validité du protocole et atteindre toutes les destinations quelle que soit la fonction de routage point-à-point sous-jacente. Les copies inutiles qui restent sont la cause de l'application de la méthode d'élimination au niveau du lien en émission plutôt qu'au niveau du noeud. En effet, un noeud peut recevoir un même message par plusieurs de ses liens, en provenance de différents voisins, et comme ceux-ci n'ont pas connaissance de cette situation il ne peuvent que dupliquer inutilement le message.

Nous pouvons de plus appliquer une élimination en réception des copies inutiles dans le cas où un message (*id*,*o*) arrive à un noeud par un lien *e* qu'il a déjà emprunté, c'est-à-dire lorsque *id* est strictement inférieur à *idsuiv*[*e*,*o*]. Cette règle permet d'éviter, lors d'une diffusion, de remonter dans l'arbre de tous les chemins de la source vers toutes les destinations. Cet arbre est ainsi l'arbre de diffusion pour *o*.

Tel que nous venons d'en décrire les caractéristiques, nous pouvons maintenant donner l'algorithme de base de notre protocole de diffusion correcte pour un réseau de *N* processeurs, chacun de degré *d*. Pour le code Occam du routeur à diffusion que nous donnons ci-après, nous avons supposé l'existence de canaux spéciaux (tableau `Lien`) pour l'accès aux liens de communication du noeud:

```
[d] CHAN OF ANY Lien:
```

Ces canaux peuvent s'utiliser en réception comme en émission afin d'utiliser les liens dans les deux sens de communication. L'algorithme est donc le suivant:

```
PROTOCOL Message_routé IS  Message, INT, INT:
PROC Diffusion(
    CHAN OF Message De.Application, Vers.Application,
    INT n) -- identification du noeud

  [N] INT iddel:
  [d][N] INT idsuiv:
  Message m:
  INT i, o, l, id, id.local:
  SEQ
    id.local := 0 -- initialisations
    SEQ i = 1 FOR N
      SEQ
        iddel[i] := 0
        SEQ l = 1 FOR d
          idsuiv[l][i] := 0
  WHILE TRUE
    ALT
      ALT l = 1 FOR d
        Lien[l]?(m,o,id) -- routage
          IF
            id >= idsuiv[l][o] -- message attendu
              PAR
                idsuiv[l][o] := id + 1
                IF
                  id > iddel[o] -- message à délivrer
                    SEQ
                      Vers.Application!m
                      iddel[o] := id
                  TRUE
                    SKIP
                PAR i = 1 FOR d
                  IF
                    i∈D_n[l] and id >= idsuiv[l][o]
                      SEQ
                        Lien[i]!(m,o,id)
                        idsuiv[i][o] := id + 1
                    TRUE
                      SKIP
            TRUE
              SKIP
      De.Application?m -- injection
        SEQ
          PAR i = 1 FOR d
            SEQ
              Lien[i]!(m,n,id.local)
              idsuiv[i][n] := id.local + 1
          id.local := id.local + 1
:
```

5.2.2 Correction de l'algorithme

Pour vérifier que notre algorithme est correct il nous faut démontrer sa ***terminaison***, sa ***validité***, et qu'il introduit ***pas d'interblocage***. L'*équité* et l'*absence de famine* sont, tout comme pour le routage point-à-point, garanties par l'utilisation de files de messages de type *premier entré/premier sorti* pour chaque lien.

Terminaison

Supposons qu'une diffusion ne se termine pas, c'est-à-dire que le message reste continuellement dans le réseau. Alors à tout instant il existe au moins une copie du message estampillée (*id*,*o*), arrivée à un noeud *n* par un lien *e*, qui ne sera pas éliminée; cette copie sera donc transmise sur l'ensemble des liens *s* qui vérifient:

$$s \in D_n[e] \wedge id \geq idsuiv[s, o]$$

Il faut donc que le message n'ait pas déjà emprunté le lien *e* pour atteindre *n*, auquel cas la nouvelle copie aurait été ignorée car tous les liens de $D_n[e]$ ne pourraient de nouveau laisser transiter le message:

$$\forall s \in D_n[e], idsuiv[s, o] \geq id + 1$$

Comme le nombre de liens du réseau est fini, et puisque l'algorithme élimine toute duplication sur un lien, au bout d'un temps fini l'ensemble des copies d'un message diffusé seront éliminés, faute de liens non encore empruntés; d'où la terminaison de la diffusion.

Validité

Pour démontrer la validité de notre algorithme nous allons montrer que la diffusion préserve un *ordre partiel* relatif à la source, et qu'elle est *atomique*. Il s'agit donc en premier lieu de prouver que pour toute source, tous les messages envoyés sont reçus par toutes les destinations dans l'ordre suivant lequel ils ont été diffusés.

Sous l'hypothèse de fiabilité des noeuds et du réseau qui est couramment admise pour les architectures parallèles, lorsque la fonction de routage point-à-point est valide, au sens où pour toute paire de processeurs elle définit au moins un chemin, notre algorithme permet d'atteindre toutes les destinations quelle que soit la source. En effet, les tables de diffusion sont calculées à partir des tables de routage point-à-point de manière à ne changer aucun chemin valide entre noeuds.

Il faut cependant garantir qu'un site n'ignore pas un message qu'il n'a pas encore délivré et rediffusé. Ce cas ne peut se produire que s'il reçoit les messages d'une même source dans le désordre. Cela se caractérise de la manière suivante:

- soit *s* un site qui diffuse les messages m_1 puis m_2, de numéros respectifs id_1 et id_2, avec $id_2 = id_1 + 1$;
- le noeud *i* reçoit m_2 puis m_1.

Dans cette situation le site *i*, lors de la réception de m_2, attendait le message m_1 de numéro $id_1 = iddel[o] + 1$; comme $id_2 = id_1 + 1 \geq iddel[o]$ ce message est délivré et $iddel[o] = id_2$, c'est-à-dire que le prochain message attendu est celui de numéro $iddel[o] + 1 = id_2 + 1$. Ainsi lorsque *i* reçoit m_1, il ne délivre pas ce message car $id_1 < iddel[o]$.

Notons d'ailleurs que cette élimination de m_1 est répercutée également sur les liens que les deux messages auraient normalement dus empruntés: si *l* est un de ces liens alors au passage de m_2, $idsuiv[l,o]$, qui était égal à id_1 donc inférieur à id_2, prend la valeur de $id_2 + 1$; ainsi lorsque m_1 devrait être retransmis sur *l* la valeur de $idsuiv[l,o]$ l'en empêche car $id_1 < idsuiv[l,o]$.

Dans un tel cas de figure il existe un site *j*, précédent *i* dans l'arbre des chemins suivis par m_1 et m_2, pour lequel la réception des deux messages se fait dans l'ordre correct: m1 puis m2, mais à partir duquel m_1 et m_2 suivent deux routes R_1 et R_2 différentes. Et du fait de trafics ou de longueurs de routes différents, le message m_2 «*double*» m_1 et arrive en premier au site *i*, comme l'illustre la figure 5.1 ci-après.

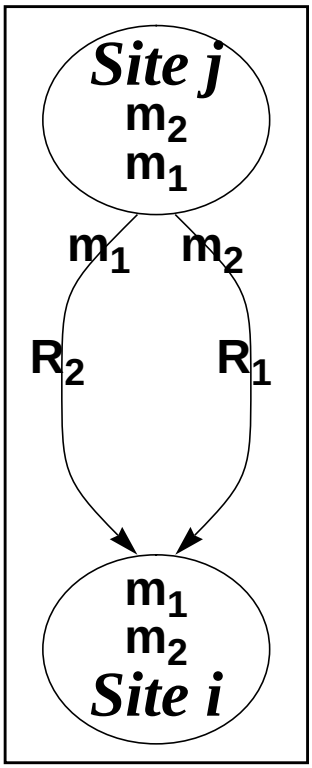
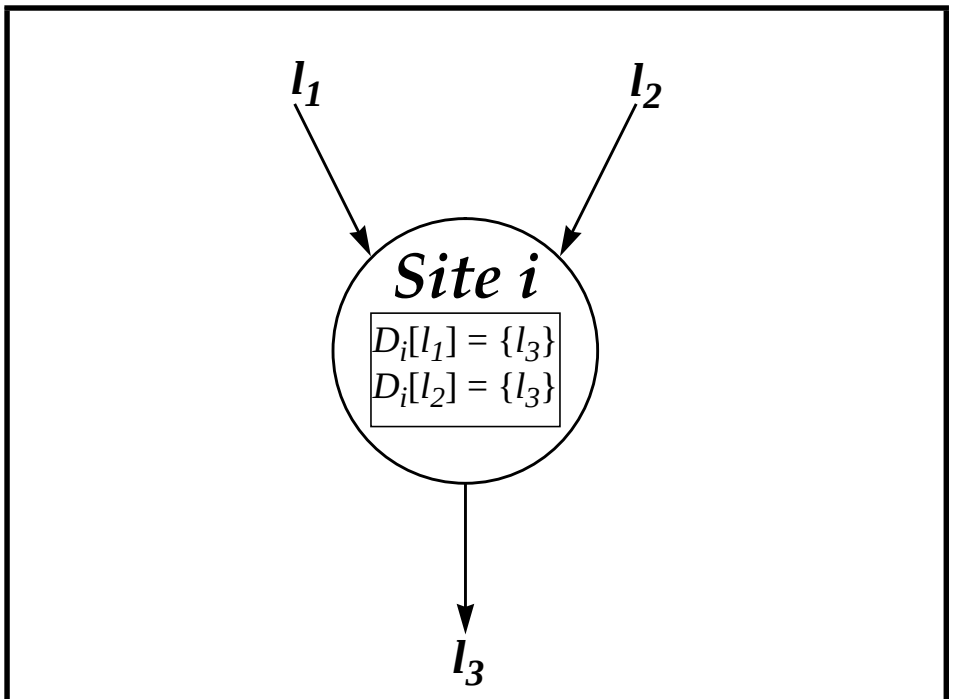


Figure 5.1 : réception non ordonnée de messages sur un site.

Pour que ces deux messages suivent des routes différentes à partir du site *j*, il faut que leurs premières réceptions se fasse par des liens différents, autrement ils seraient rediffusés sur les mêmes liens *s* qui vérifient:

$$s \in D_n[e] \wedge id \geq idsuiv[s, o]$$

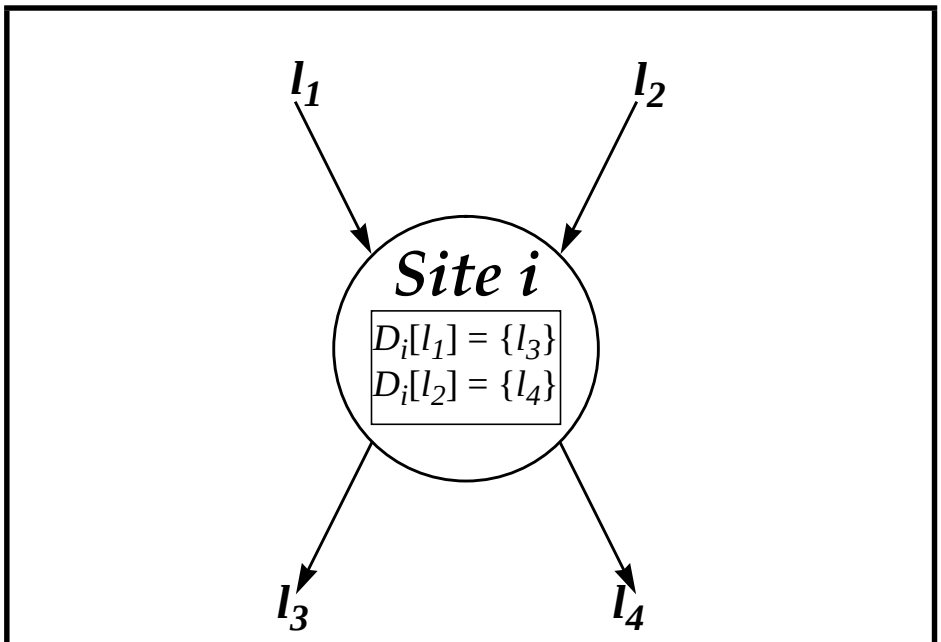
où *e* est le lien d'arrivé. Ils suivraient alors les mêmes routes, et ainsi arriveraient dans le bon ordre au site *i*. Nous nous placerons donc dans la situation où les messages arrivent par des liens différents.



	Ordre des arrivées		Ordre des sorties sur l_3	
	sur l_1	sur l_2		
1	m_1		m_1	↓ temps
2	m_2		m_2	
3		m_1		
4		m_2		
1	m_1		m_1	↓ temps
2		m_1		
3	m_2		m_2	
4		m_2		
1	m_1		m_1	↓ temps
2		m_1		
3		m_2	m_2	
4	m_2			

Figure 5.2 : rediffusion sur les mêmes liens quel que soit le lien d'arrivée.

Deux cas se présentent alors: les liens de sorties autorisés sont identiques pour les deux liens d'entrée - pour simplifier, et sans perte de généralité, nous ne considéreront qu'un seul lien -, ou les liens de sorties autorisés diffèrent selon le lien d'arrivée - là encore nous réduirons la situation à un seul lien de sortie par lien d'entrée. Dans le premier cas, comme le montre la figure 5.2, tous les ordres de réception des messages produisent le même ordre de rediffusion sur le lien de sortie. Il en est de même lorsque les liens de sortie diffèrent, puisqu'ils reçoivent tous deux les messages dans le bon ordre, comme l'illustre la figure 5.3.



	Ordre des arrivées		Ordre des sorties		
	sur l_1	sur l_2	sur l_3	sur l_4	
1	m_1		m_1		↓ temps
2	m_2		m_2		
3		m_1		m_1	
4		m_2		m_2	
1	m_1		m_1		↓ temps
2		m_1		m_1	
3	m_2		m_2		
4		m_2		m_2	
1	m_1		m_1		↓ temps
2		m_1		m_1	
3		m_2		m_2	
4	m_2		m_2		

Figure 5.3 : rediffusion sur des liens différents en fonction du lien d'arrivée.

Nous avons ainsi démontré que tout noeud du réseau préserve sur chacun de ses liens l'ordre de diffusion des messages pour toute source. Donc notre protocole maintient un ordre partiel, et par conséquent il est atomique au sens où tout site reçoit et délivre exactement une fois chaque message.

Absence d'interblocage

Comme nous l'avons vu au chapitre 3, une fonction de routage *f* est sans interblocage lorsqu'elle n'introduit pas de cycle dans le graphe de dépendances entre liens. Dans un tel graphe chaque sommet correspond à un lien du réseau, et chaque arc ($l_1 \rightarrow l_2$) est une dépendance entre les deux liens l_1 et l_2 dans les tables de routage *R* de *f* (selon les notations introduites au chapitre 3 §3.2.1), i. e.[1]:

$$\exists (i, j) \in P \times P, l_2 \in R_i [l_1, j]$$

Il nous faut maintenant démontrer que notre algorithme de diffusion préserve la structure acyclique du graphe de dépendances. Or, en tout noeud *n*, la retransmission d'un message parvenu par le lien *e* ne se fait que sur les liens de l'ensemble $D_n[e]$. Comme:

$$\forall (n , e) \in P \times L, D_n [e] = \{ s, \exists d \in P, s \in R_n [e, d] \}$$

la rediffusion respecte toutes les contraintes imposées par la fonction de routage et n'introduit aucune dépendance supplémentaire. Il est donc clair que notre protocole n'induit pas de situation d'interblocage lorsque la fonction de routage, au-dessus de laquelle il est construit, est elle-même sans interblocage.

Nous avons donc montré que notre algorithme de diffusion est correct, c'est-à-dire qu'il se termine en un temps fini, qu'il est valide et sans interblocage.

5.2.3 Analyse des performances de l'algorithme

Contrairement aux techniques que nous avons présentées au chapitre précédent, notre méthode ne permet pas aussi aisément de calculer les performances de l'algorithme, en termes d'encombrement et de profondeur d'arbre de diffusion. En fait, comme nous allons le mettre en évidence dans cette section, celles-ci sont fortement dépendantes de la fonction de routage employée. Pour illustrer ce phénomène, et donner une évaluation de l'algorithme selon les deux critères de performances que nous nous sommes fixés, nous allons donner plusieurs exemples de diffusions avec les deux fonctions de routage disponibles dans ParX: celle construite à partir d'un arbre recouvrant et celle qui repose sur un cycle eulérien.

Le réseau que nous avons retenu se compose d'une grille torique carrée de 16 processeurs modifiée pour accueillir un noeud supplémentaire, tel que cela est représenté dans la figure 5.4. Ce processeur additionnel permet d'une part de contrôler le réseau et/ou l'application, et d'autre part de réaliser la connexion du réseau à une machine hôte. Une telle configuration est fréquente, notamment avec les machines Supernodes qui nous ont servi de support d'évaluation.

1. on obtient des relations similaires avec des tables inversées, qui tiennent ou non compte du lien d'entrée.

De ce réseau nous pouvons extraire les graphes recouvrants qui sont utilisés pour calculer les deux fonctions de routage au-dessus desquelles nous appliquerons notre algorithme de diffusion. Il s'agit tout d'abord du cycle eulérien de la figure 5.4, dont un lien reste inutilisé.

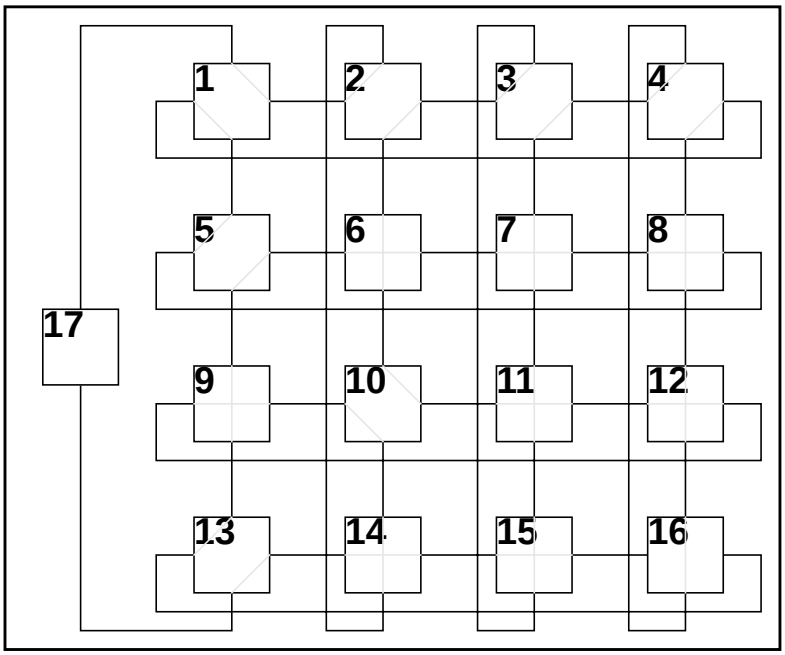


Figure 5.4 : tore4x4 avec hôte et un cycle eulérien.

Le second graphe recouvrant est l'arbre qui est celui de la figure 5.5a; le schéma 5.5b fait apparaître l'arbre dans le réseau initial, avec en pointillés tous les «*extra-liens*», soit les liens du réseau qui n'appartiennent pas à l'arbre et qui interviennent dans divers chemins.

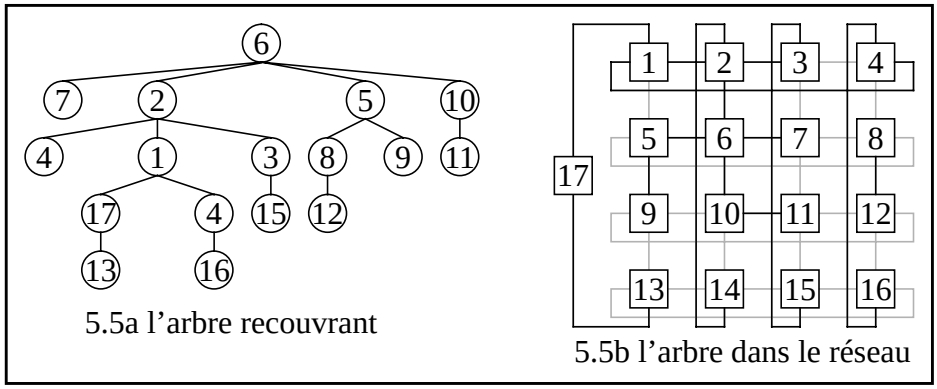


Figure 5.5 : un arbre recouvrant du tore 4x4 modifié.

A partir de ces fonctions sont calculées les divers tables de routage et de diffusion qui interviennent dans les routeurs de chaque noeud. Ce sont ces tables que nous avons utilisées pour produire nos exemples en reproduisant le comportement de notre algorithme, dans la situation où il n'y a aucune autre communication en cours. Certes il s'agit du cas de figure le plus simple et le plus favorable, mais c'est d'une part comme cela que nous avons évalué les autres techniques de diffusion; nous sommes donc dans un cas qui nous permettra de comparer nos résultats. Et d'autre part, effectuer des évaluations dans des cas plus complexes de concurrence, où vis-à-vis du comportement de certaines applications, est très difficile a exprimé mathématiquement ou dans un modèle de simulation fiable. Ce travail ne fait pas partie des objectifs de cette thèse.

Les deux premiers exemples sont issus d'une diffusion depuis le processeur 1. Nous avons obtenu l'arbre de la figure 5.6a avec la fonction de routage calculée à partir du cycle eulérien, et l'arbre de la figure 5.6b avec la fonction de routage déduite de l'arbre recouvrant. Sur ces deux exemples nous pouvons tout d'abord remarquer que la retransmission se fait en fonction du lien d'arrivée, comme c'est le cas, entre autres, du processeur 15 dans la figure 5.6a. Nous observons également qu'un même site peut être atteint par plusieurs liens (cf. processeur 11 dans la figure 5.6b), et donc par des chemins différents. Cette caractéristique est intéressante puisque dans le cas de congestions localisées à certains chemins, le message peut tout de même parvenir rapidement à ses destinations par des routes «secondaires»; nous pouvons donc dire que notre algorithme a un comportement *adaptatif*.

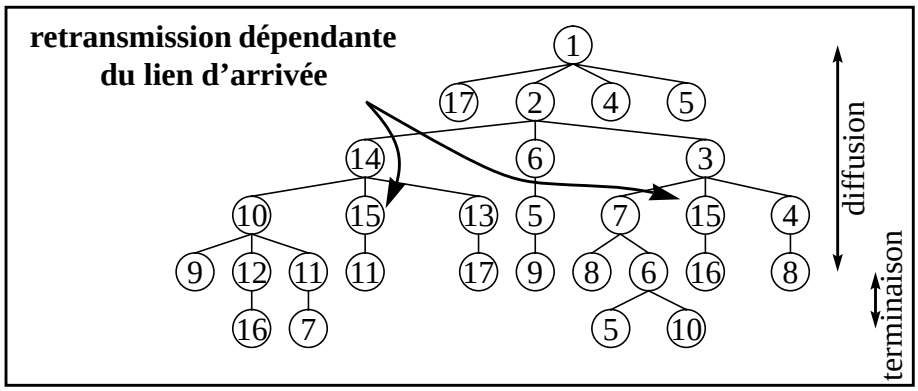


Figure 5.6a : diffusion selon le cycle eulérien depuis le noeud 1.

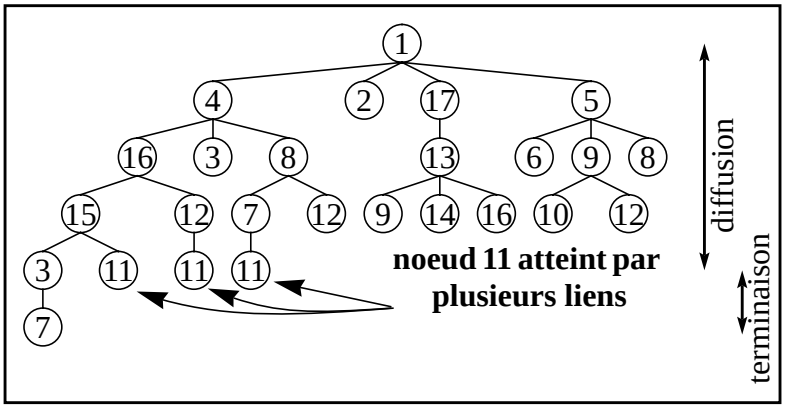


Figure 5.6b : diffusion selon l'arbre recouvrant depuis le noeud 1.

Si nous comparons le comportement du protocole avec les deux fonctions de routage, nous constatons en premier lieu que tous deux atteignent toutes les destinations en quatre étapes et qu'une étape de plus est nécessaire pour que la diffusion se termine; c'est-à-dire que leurs profondeurs d'arbre sont égales à 5. La différence se fait sur l'encombrement qui, sur cet exemple, est plus important avec le cycle eulérien comme graphe recouvrant qu'avec l'arbre (28 liens au lieu de 25). Toutefois, l'utilisation des liens dans le cas du cycle eulérien comme structure de calcul des tables, semble légèrement mieux répartie dans l'arbre de diffusion. En effet pour l'arbre 5.6b, le nombre de noeuds est plus important dans les étapes initiales que dans les étapes finales. Ainsi l'encombrement induit avec le cycle eulérien est mieux distribué tout au long de la diffusion, ce qui évite d'aboutir à des situations de forte congestion préjudiciables pour l'ensemble des communications effectuées dans ces instants.

Les deux diffusions suivantes, toutes deux de source le processeur 14, vont nous permettre de mieux apprécier la différence de comportement de l'algorithme selon la fonction de routage considérée. Comme nous le voyons sur les deux figures 5.7a et 5.7b, cette fois-ci le cycle eulérien utilisé apparait être un graphe recouvrant qui induit un encombrement moindre: seulement 26 liens sont nécessaires, alors qu'avec l'arbre choisi il faut 33 liens, soit la totalité. En revanche ceci se fait au détriment de la profondeur de l'arbre de diffusion, où il faut plus d'étapes, avec le cycle eulérien pour atteindre toutes les destinations. Ceci est dû à une répartition plus équitable des noeuds dans les différentes étapes lorsqu'un cycle eulérien est utilisé. Inversement, si la diffusion est effectuée avec l'arbre recouvrant comme fonction de routage, l'arbre de diffusion obtenu est plus large, c'est-à-dire que plus de processeurs sont atteints dès les premières étapes. La phase de diffusion est donc globalement plus courte et plus encombrée avec l'arbre recouvrant comme fonction de routage point-à-point sous-jacente. Le cycle eulérien quant à lui offre une communication plus étalée dans le temps et évite des périodes de plus fort encombrement.

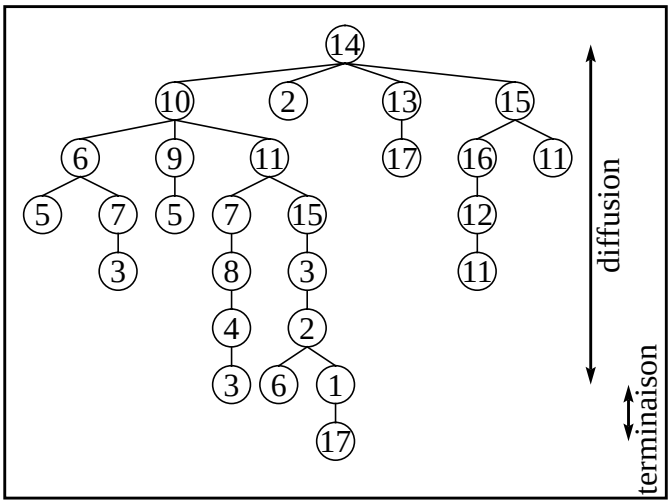


Figure 5.7a : diffusion selon le cycle eulérien depuis le noeud 14.

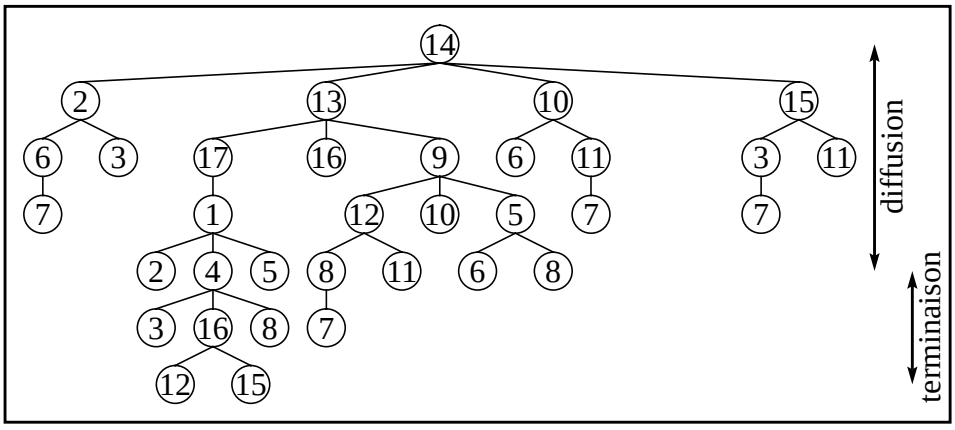


Figure 5.7b : diffusion selon l'arbre recouvrant depuis le noeud 14.

En somme, notre algorithme d'inondation, en plus de respecter les contraintes de la fonction de routage sous-jacente et de garantir ainsi l'absence d'interblocage, présente des caractéristiques intéressantes pour son utilisation dans les architectures massivement parallèles. Tout d'abord, sous la condition que la fonction de routage autorise plusieurs chemins entre deux noeuds, il s'adapte aux conditions dynamiques du trafic dans un réseau. Enfin, et surtout, il est possible

d'influer sur le comportement des diffusions en changeant de fonction de routage. En effet, comme nous venons de le mettre en évidence, certaines fonctions produisent un encombrement réduit pour un arbre de diffusion plus profond, alors qu'inversement d'autres induisent un trafic plus important en minimisant la profondeur de l'arbre. Il est important de noter que dans le cas particuliers des réseaux réguliers, en utilisant des tables de routage pré-calculées selon un algorithme optimal (de type e-cube), nous obtenons une diffusion minimale, tant pour l'encombrement que pour la profondeur de l'arbre de diffusion.

5.3 Utilisation du routage par intervalles pour la diffusion

Le compactage de l'information de routage sous forme d'intervalles, qui associent à tout lien de communication l'ensemble des destinations qu'il permet de rejoindre, ne permet plus à notre algorithme, tel qu'il est décrit section 5.2.1 de fonctionner. Cela vient du fait qu'avec un tel étiquetage des liens les dépendances, interdites ou autorisés, ne sont plus explicites. Il n'est alors plus possible de calculer les tables de diffusion, et notre protocole se ramène à une inondation classique. En effet, dans ce cas, lorsqu'un message arrive à un noeud par un lien, il ne peut qu'être retransmis sur tous les autres liens pour garantir la validité de la diffusion.

Une réponse pour l'accès à ces dépendances consiste à mémoriser dans l'en-tête de chaque message les processeurs déjà atteints et ceux qui n'ont pas encore reçu le message. Nous avons alors besoin d'un bit par noeud, ce qui signifie un champ de taille proportionnelle au nombre de processeurs du réseau. L'inconvénient de cette méthode est qu'elle ne tire pas partie de l'étiquetage par intervalles pour réduire la taille des messages.

Notre solution est de ne conserver dans l'en-tête que les processeurs qui restent à atteindre, et de représenter cet ensemble dans un intervalle. Ainsi lorsqu'un message arrive en un noeud, il ne sera retransmis que sur les liens pour lesquels l'intersection de l'intervalle du lien et celui du message n'est pas vide. Sur chacun de ces liens l'intervalle associé au message est remplacé par cette intersection. Avec une telle méthode nous retrouvons, pour la diffusion, la notion de correction appliquée au routage par intervalles. Il ne s'agit donc pas d'une technique de diffusion nouvelle, mais simplement d'une adaptation de notre algorithme aux contraintes du routage par intervalles.

De plus, comme les fonctions de calcul des intervalles produisent le plus souvent un routage déterministe, l'élimination des copies superflues devient alors inutile. Nous obtenons alors le nouvel algorithme suivant:

```
PROTOCOL Message_routé IS Interval, Message:

PROC Diffusion.interval(
    CHAN OF Message De.Application, Vers.Application,
    [d] CHAN OF Message_routé De.voisin, Vers.voisin,
    [d] Interval Etiq.lien)

  Message m:
  Interval Etiq.m:
  INT i, j:

  WHILE TRUE
    ALT
      ALT i = 0 FOR d
```

```
        Lien[i]?(Etiq.m,m)
          PAR
            Vers.Application!m
            PAR j = 0 FOR d
              IF
                i<>j and Etiq.lien[j]∩Etiq.m <> ∅
                  Lien[j]!(Etiq.lien[j]∩Etiq.m,m)
                TRUE
                  SKIP

      De.Application?m
        PAR i = 0 FOR d
          Lien[i]!(Etiq.lien[i],m)
```

:

Il faut cependant restreindre l'utilisation de cet algorithme aux schémas d'étiquetage qui produisent des intervalles linéaires [BLT91], où pour tout intervalle [*a*,*b*) *b* est supérieur ou égal à *a*. En effet, l'intersection d'un intervalle [*a*,*b*) avec $b < a$ avec un intervalle [*c*,*d*) où $d \geq c$ peut conduire à deux intervalles; et par voie de conséquence, au fur et à mesure des intersections à un nombre toujours plus grand d'intervalles. Par exemple, pour un réseau de 100 processeurs, l'intersection de [70,10) avec [5,80) est égale aux deux intervalles [5,10) et [70,80).

Malgré ces limitations, parce que beaucoup de schémas d'étiquetage respectent cette contrainte, et notamment ceux sur les réseaux réguliers, notre algorithme reste grandement utilisable dans les architectures parallèles. Il pourrait par ailleurs s'appliquer aux schémas d'étiquetage étendus définies dans [Mug93], sous l'hypothèse toutefois d'intervalles toujours linéaires.

Enfin, la possibilité de ne diffuser un message que vers un sous-ensemble des processeurs, et non plus à tous, est un atout non négligeable de notre solution. Il suffit pour cela que l'application associe à chaque message l'intervalle des destinations; c'est la succession des intersections qui permettra de limiter les noeuds atteints et l'encombrement total.

5.4 Intégration de la diffusion dans un circuit routeur

Notre algorithme de diffusion a été intégré au routeur de ParX dans la réalisation du prototype pour les architectures à base de transputers. Il s'agit d'un routeur logiciel de type *store-and-forward*, qui peut tout à fait être découplé du processeur dans un circuit dédié. Toutefois une telle réalisation d'un composant matériel serait beaucoup plus efficace avec des techniques d'acheminement de types ***wormhole*** ou ***virtual-cut-through***.

Avec de telles méthodes le routage par intervalles s'avère grandement profitable, notamment au niveau de la complexité du circuit, et par conséquent sur le délai de traversé des messages dans un routeur. Par contre cela nécessite l'emploi d'un arbitrage des liens adapté à la fois aux échanges point-à-point et aux diffusions.

L'architecture d'un tel routeur, comme nous pouvons le voir sur la figure 5.8, est donc constituée, en plus des liens et d'un circuit de commutation, d'une interface de traitement des messages pour chaque lien d'entrée, d'un arbitre par lien de sortie[1] et deux voies de signalisation

1. nous prenons ici l'option d'un arbitrage réparti plutôt que centralisé afin d'optimiser cette phase.

entre toute interface et tout arbitre. La première voie, nommée *Requête*, sert à l'interface d'une entrée à signaler une requête de réservation de la sortie; elle est maintenue jusqu'à ce que l'accès soit obtenu et lors de la retransmission, après quoi l'interface relâche le signal pour libérer le lien de sortie. La seconde voie, nommée *Accès,* permet à l'arbitre d'autoriser l'accès à l'interface; le signal est maintenu tant que la voie *Requête* l'est aussi.

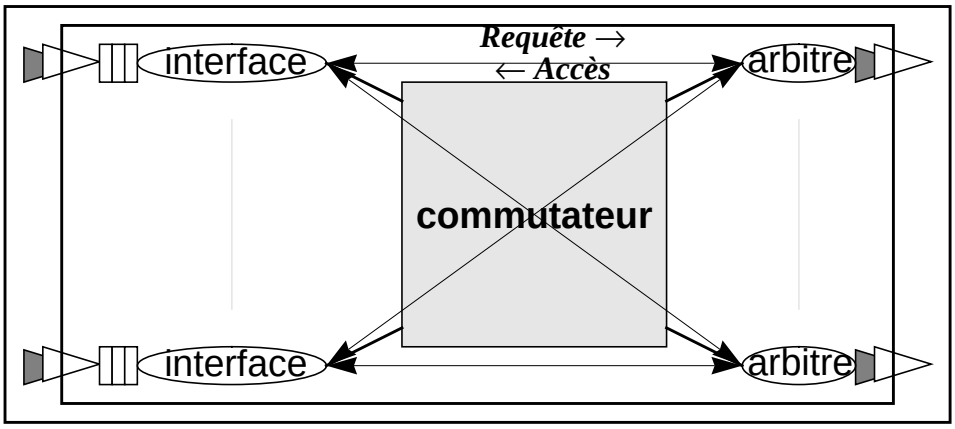


Figure 5.8 : architecture matérielle du routeur à diffusion.

L'arbitrage intervient lorsque plusieurs messages arrivent de différents liens et doivent transiter par le même lien de sortie. Il faut alors résoudre les conflits d'accès et garantir une sélection la plus équitable possible afin de ne pas générer de famine, ou de congestion si le trafic le permet. Les arbitres à priorités circulaires, implémentés comme des automates d'états finis, sont des solutions peu coûteuses et efficaces pour réaliser des allocations équitables et rapides dans le matériel. Dans ces arbitres les priorités respectives des liens d'entrée dépendent de la dernière sélection placée dans le registre d'état de l'automate.

Lorsqu'un message parvient à un routeur, l'interface du lien d'entrée décode l'en-tête et lance un signal de requête aux arbitres de tous les liens de sorties données par la fonction de routage point-à-point ou l'algorithme de diffusion. Une fois l'accès obtenu, le message est retransmis sur le lien de sortie et le signal de réservation relâché pour libérer le lien de sortie. Dans l'algorithme d'un arbitre exprimé ci-dessous, nous avons introduit le type *voie de signalisation* pour lequel nous disposons des opérateurs suivants: **!** pour émettre en continue un signal, **¡** pour stopper l'émission d'un signal, **?** pour attendre un signal, **¿** pour attendre sa suspension et **~** pour tester un signal.

```
WHILE TRUE
  SEQ
    entrée := (courant + 1) MODULO nb_entrées
    sélection := AUCUN_LIEN
    WHILE sélection = AUCUN_LIEN
      IF
        Requête[entrée]~
          sélection := entrée
        TRUE
          entrée := (entrée + 1) MODULO nb_entrées
    courant := sélection
    Accès[courant]!
    Requête[courant]¿
    Accès[courant]¡
```

Associée à l'arbitrage, une politique de gestion des liens de sorties doit être définie pour déterminer si un message peut ou non demander l'accès à plusieurs liens de sorties et être autorisé à transmettre simultanément sur ces liens. Nous avons considéré quatre politiques: la première est séquentielle et déterministe, la seconde séquentielle et adaptative, la troisième parallèle et asynchrone et la dernière parallèle et synchrone. Outre leurs algorithmes, nous examinerons leurs répercussions sur la complexité de l'arbitrage, les besoins en tampons, et la latence de diffusion.

Nos deux politiques séquentielles traitent la diffusion comme une séquence de transmissions du même message sur un lien. Ces politiques, comme la troisième, nécessitent de stocker entièrement le message et interdisent donc de ce fait l'utilisation du wormhole ou du virtual-cut-through.

5.4.1 Politique séquentielle déterministe

Dans cette politique, tout message ne peut simplement demander et avoir accès qu'à un seul lien à la fois. L'arbitrage est alors simple et le délai de sélection du lien d'entrée ne requiert qu'un seul cycle d'horloge. Dans le cas d'une diffusion, tout message suit une séquence prédéterminée d'accès aux liens. Malheureusement, cette séquence ne correspond pas forcément à l'ordre dans lequel les liens deviennent libres; ce qui implique une mauvaise latence de diffusion.

L'algorithme de l'interface d'un lien d'entrée (identifiée par la variable locale `entrée`) peut être explicité comme suit, où `d` est le degré du noeud:

```
WHILE TRUE
  SEQ
    Lien[entrée]?(en-tête,message)

    IF
      diffusion(en-tête) -- est-ce un message diffusé?
        SEQ sortie = 0 FOR d -- séquence prédéfinie
          IF
            sortie ∈ Dn[entrée]
              SEQ
                Requête[sortie]!
                Accès[sortie]?
                -- s'il y a lieu modification de l'en-tête
                Lien[sortie]!(en-tête,message)
                Requête[sortie]¡
            TRUE
              SKIP

      TRUE -- échange point-à-point
        SEQ
          sortie := Rn[entrée][destination(en-tête)]
          Requête[sortie]!
          Accès[sortie]?
          Lien[sortie]!(en-tête,message)
          Requête[sortie]¡
```

5.4.2 Politique séquentielle adaptative

Il s'agit d'un algorithme où les requêtes sont émises en parallèle, et la retransmission se fait sur le premier lien de sortie disponible, i. e. celui dont on reçoit le signal d'accès. Avec cette stratégie la diffusion s'opère ainsi selon l'ordre optimal d'accès aux liens de sortie.

A la fin d'un transfert de message, l'arbitre donne l'accès au message de plus haute priorité, priorité qui est celle de l'interface dans laquelle il se trouve. Comme plusieurs autorisations d'accès peuvent être accordées au même message, tous les signaux de requête, excepté celui qui correspond au lien par lequel le message sera envoyé, doivent alors être relâchés. Pour chacune de ces libérations de lien un nouvel arbitrage doit être fait et pendant ce temps le lien reste inutilisé. Dans le pire des cas, l'arbitrage d'un lien peut demander autant de cycles qu'il y a de liens d'entrée.

L'algorithme de l'interface d'un lien d'entrée est donné ci-après. Nous pouvons remarquer qu'une diffusion correspond à la répétition, pour le même message, du routage point-à-point.

```
WHILE TRUE
  SEQ
    Lien[entrée]?(en-tête,message)

    IF
      diffusion(en-tête)
        SEQ
          sorties := Dn[entrée]
          WHILE sorties <> ∅
            SEQ
              PAR l = 0 FOR d -- requêtes en parallèle
                IF
                  l ∈ sorties
                    Requête[l]!
                  TRUE
                    SKIP
              ALT l = 0 FOR d -- attente du premier accès
                l ∈ sorties & Accès[l]?
                  SEQ
                    PAR r = 0 FOR d -- libérations
                      IF
                        l ∈ sorties and r <> l
                          Requête[r]¡
                        TRUE
                          SKIP
                    -- modification en-tête éventuelle
                    Lien[l]!(en-tête,message)
                    Requête[l]¡

      TRUE -- routage point-à-point
        SEQ
          sorties := Rn[entrée][destination(en-tête)]
          PAR l = 0 FOR d -- requêtes en parallèles
```

```
            IF
              l ∈ sorties
                Requête[l]!
              TRUE
                SKIP
          ALT l = 0 FOR d -- attente premier accès
            l ∈ sorties & Accès[l]?
              SEQ

                PAR r = 0 FOR d -- libérations
                  IF
                    l ∈ sorties and r <> l
                      Requête[r]¡
                    TRUE
                      SKIP
                Lien[l]!(en-tête,message)
                Requête[l]¡
```

5.4.3 Politique parallèle asynchrone

Cette fois-ci, l'interface d'entrée émet toutes les requêtes de sorties en parallèle, et traite les réponses d'accès des arbitres également en parallèle. Ainsi, aussitôt qu'un accès à une sortie est obtenu, le message est retransmis sur le lien, puis le lien libéré. L'interface ne se met attente du prochain message que lorsque le dernier transfert a été effectué.

Comme aucune requête n'est éliminée sans transfert, le délai d'arbitrage est ramené à un seul cycle, si bien que cette méthode produit une latence minimale aussi bien pour le routage point-à-point que pour la diffusion.

L'inconvénient de cette politique réside dans l'utilisation de la mémoire de stockage du message. En effet, l'accès à un tampon d'entrée doit pouvoir se faire en parallèle, afin de ne pas produire de goulot d'étranglement à ce niveau. Pour cela une mémoire multi-ports s'avère nécessaire. De tels circuits sont d'autant plus complexes qu'ils possèdent de ports, si bien que leur utilisation réduirait notablement le nombre de liens du routeur.

Dans l'algorithme ci-dessous, la partie point-à-point n'est pas détaillée; il s'agit en fait du même algorithme que pour la politique séquentielle déterministe.

```
WHILE TRUE
  SEQ
    Lien[entrée]?(en-tête,message)

    IF
      diffusion(en-tête)
        PAR sortie = 0 FOR d
          IF
            sortie ∈ Dn[entrée]
              SEQ
                Requête[sortie]!
                Accès[sortie]?
                -- modification éventuelle de l'en-tête
```

```
                Lien[sortie]!(en-tête,message)
                Requête[sortie]¡
            TRUE
              SKIP

      TRUE
        -- routage point-à-point
```

5.4.4 Politique parallèle synchrone

Dans le mode parallèle synchrone, le message est retransmis en parallèle sur toutes les sorties une fois que l'accès a été obtenu de tous les arbitres. Ainsi tout message diffusé requiert un accès simultané et exclusif à tous les liens de sortie pour être rediffusé. La libération des liens se fait également en parallèle.

Comme les liens sont des ressources à accès exclusif, des interblocages peuvent se produire si l'arbitrage ne prend pas cela en compte. Par exemple, prenons deux messages m_1 et m_2 qui doivent être retransmis sur la même paire de liens (l_1,l_2). Si nous supposons maintenant qu'au premier cycle d'arbitrage m_1 reçoive l'accès à l_1 et m_2 à l_2, alors chaque message attendra indéfiniment le lien alloué à son concurrent; et aucun des deux ne pourra continuer sa progression.

Pour éviter qu'une telle situation ne se produise, il est nécessaire de respecter un ordre d'allocation des ressources pour éviter les interblocages. Un message ne peut alors émettre une requête vers une sortie que lorsqu'il a obtenu l'autorisation d'accès aux liens de sorties précédents. Dans notre exemple, cette règle empêche m_2 d'obtenir l'accès l_2 alors que m_1 reçoit son autorisation pour l_1; ainsi, l'accès de m_1 à l_2 est possible, et sa progression peut donc se faire.

Bien évidemment, cette méthode peut laisser les liens inutilement inactifs. En effet, dans l'ensemble des liens de sorties que doit emprunter un message, ceux déjà alloués peuvent rester inusités durant le temps où les autres (de rang supérieur) sont occupés par d'autres messages.

Par contre, le problème d'accès concurrents au tampon ne se pose plus. De plus, puisque la rediffusion ne se fait qu'en une seule fois, il est possible d'utiliser les techniques du wormhole et du virtual-cut-through. Avec le wormhole le tampon peut être réduit au minimum et le stockage réalisé par les noeuds précédents. Toutefois, parce qu'une taille de tampon de cet ordre impliquerait un grand nombre de liens bloqués et oisifs (la quasi-totalité des liens dans tous les noeuds atteints jusqu'au moment du blocage), le virtual-cut-through semble meilleur.

L'algorithme d'une interface d'entrée est le suivant:

```
WHILE TRUE
  SEQ
    Lien[entrée]?(en-tête,message)

    IF
      diffusion(en-tête)
        SEQ
          FOR sortie = 0 FOR d -- ordre d'allocation
            IF
              sortie ∈ Dn[entrée]
```

```
            SEQ
              Requête[sortie]!
              Accès[sortie]?
          TRUE
            SKIP
      -- modification éventuelle de l'en-tête

    PAR sortie = 0 FOR d -- rediffusion
      IF
        sortie ∈ Dn[entrée]
          Lien[sortie]!(en-tête,message)
        TRUE
          SKIP
    PAR sortie = 0 FOR d -- libération
      IF
        sortie ∈ Dn[entrée]
          Requête[sortie]¡
        TRUE
          SKIP

TRUE
  -- routage point-à-point
```

5.4.5 Conclusion

Pour la réalisation d'un routeur matériel qui intègre à la fois le routage point-à-point et la diffusion nous avons proposé, dans ce chapitre, différents algorithmes selon le type de fonction de routage, la technique d'acheminement ou la représentation des tables de routage. Tous ces algorithmes se basent sur une solution correcte au routage pour la diffusion dans le contexte des architectures massivement parallèles.

Notre but n'étant pas, dans un premier temps, la réalisation d'un tel circuit routeur, nous ne préconisons pas une implémentation par rapport à une autre. Un tel choix ne peut se faire que pour obtenir un bon compromis entre la taille résultante du circuit, la latence de retransmission d'un noeud, la taille mémoire nécessaire, ou sur d'autres paramètres encore.

Partie III : La Machine Virtuelle à Diffusion

Le protocole que nous venons de décrire rend possible la diffusion de messages dans tout réseau de processeurs. Mais il n'est accessible qu'au niveau des processeurs, son utilisation doit donc être étendue maintenant à l'ensemble des processus d'une application. Or pour l'application, et pour son concepteur, la diffusion doit avant tout être un mode de communication entre les processus qui la compose. Comme il n'est pas plus concevable de remettre à la charge de l'utilisateur la définition et l'implémentation des protocoles de communications globales pour ses processus, que de le laisser réaliser le routage, tout système de communication à diffusion ne peut se contenter d'un mécanisme de diffusion inter-processeur; il doit donc offrir des protocoles accessibles aux processus.

Evidemment disposer d'un routeur à diffusion, tel que celui que nous venons de présenter au chapitre précédent, permet de construire plus aisément et plus efficacement des protocoles au niveau des processus application. En effet, la présence d'un routeur de ce type nous dégage de la tâche difficile du routage, et il ne reste plus qu'à se préoccuper de la synchronisation et de la coopération entre les processus. Cette partie traite de la construction d'un tel système et montre bien que la classe des environnements parallèles où la diffusion tient une place primordiale ne peut réellement être conçue correctement qu'avec une décomposition en deux niveaux:

1- le routage et l'acheminement des messages qui assure la communication entre les processeurs, et qui peut éventuellement être intégré dans le matériel;

2- les protocoles qui, outre leur fonction d'échange de messages, offrent aux processus des mécanismes de contrôle d'exécution, de synchronisation et de coopération.

Notre approche dans la construction de notre système de diffusion suit parfaitement cette décomposition, autant parce que nous sommes convaincu de sa justesse et de son efficacité, que parce que cela s'intègre parfaitement dans l'architecture de PAROS/ParX. Après avoir défini un routeur à diffusion, nous avons étudié et réalisé une machine virtuelle à partir de laquelle divers systèmes à diffusion peuvent être intégrés au système. C'est cette machine que nous allons présenter dans cette troisième partie.

Nous avons déjà présenté, dans les chapitres 1 et 2, les modèles de programmation qui font de la diffusion le mode de communication essentiel, ainsi que les principaux environnements de développement et bibliothèques qui proposent des fonctions de communications globales. De cette étude nous avons pu dégager et caractériser les différents protocoles qui sont nécessaires. Il nous est également apparu que la gestion des groupes de processus s'avère quasiment incontournable, et comment elle est réalisée dans ces environnements.

L'étude et l'analyse des divers protocoles et des diverses manipulations des groupes de processus nous a ainsi permis de spécifier les protocoles que notre machine virtuelle doit offrir, ainsi que le support pour la gestion des groupes qu'elle doit contenir. A partir de cela nous allons détailler la réalisation de notre machine virtuelle et son intégration dans ParX (chapitre 6).

Enfin, avant de conclure sur l'ensemble de notre travail de thèse, nous effectuerons une analyse des performances de notre machine à diffusion (chapitre 7). Les résultats que nous présenterons nous permettront de dégager des performances réelles, c'est-à-dire celles qui s'appliquent aux processus communicants, mais également de déterminer des performances temporelles chiffrées de notre solution au routage pour la diffusion.

Chapitre 6 : Réalisation de la Machine Virtuelle

Dans sa conception comme machine virtuelle, PDVM [BCD93, DeMu93b, BCD94] n'a pas pour objectif de proposer un nouveau système à diffusion, mais d'offrir un support d'implémentation dans PAROS/ParX pour les divers environnements qui existent, tels que ceux que nous avons étudiés au chapitre 2. Ainsi donc, nous avons privilégié le coeur de tels systèmes, c'est-à-dire les protocoles de communications, par rapport à la richesse de l'interface et aux nombres de primitives.

Notre machine virtuelle, dont nous allons décrire l'implémentation dans le noyau de ParX, dispose d'un protocole de diffusion synchrone qui réalise un rendez-vous entre tous les protagonistes d'une communication, et d'un protocole asynchrone qui regroupe les deux modes de communication de BSP. De plus, parce que la gestion de groupes de processus pour la diffusion est prédominante et semble s'imposer, ce qui d'ailleurs est cohérent avec le modèle de processus de ParX, notre support offre une telle fonctionnalité.

La définition de deux protocoles de diffusion, synchrone et asynchrone, répond d'une part aux besoins exprimés par les systèmes à diffusion, et d'autre part représente deux approches suivies par les applications.

Après avoir décrit le système PAROS/ParX, nous présenterons PDVM: son interface et les opérations de communication qu'elle contient. Nous spécifierons la sémantique de chacun des composants de cette machine virtuelle, et motiverons nos choix et compromis vis-à-vis des systèmes à diffusion. Puis nous décrirons la réalisation et la construction des protocoles de diffusion de PDVM dans ParX.

6.1 PAROS/ParX

Face à la grande diversité des outils informatiques, aussi bien logiciels que matériels, avoir l'ambition de proposer un système d'exploitation universel, capable de fonctionner sur tout type de machines et qui propose à l'utilisateur tous les modèles de calcul et toutes les abstractions qui existent, serait pour le moins prétentieux. Toutefois l'évolution qu'ont connue les systèmes d'exploitation ces dernières années montre cette tendance à englober en un seul système des matériels hétérogènes, et à offrir des interfaces variées et aisément interchangeables. Plus encore, des projets comme Peace (voir Annexe B) ou PAROS/ParX démontrent bien qu'il n'est plus admissible que, sous prétexte de leur complexité, la classe des ordinateurs massivement parallèles ne dispose pas d'un véritable système, digne de ce nom.

C'est dans cet esprit que le système **PAROS** a été conçu [BFF89, Lang91, BCD93, CEMW93, Gian93, Talbi93, Menn93, Elleu94]. Bien que son utilisation soit possible pour des ordinateurs plus conventionnels, PAROS est destiné plus particulièrement aux machines massivement parallèles, bâties sur le paradigme de l'échange de messages. En ce sens il s'apparente à son homologue Peace; tous deux reposent sur un micro-noyau adaptable à la fois au matériel utilisé et aux besoins des applications, offrent divers de modèles de programmation et rendent la

machine multi-applications et multi-utilisateurs. Cependant malgré leurs similitudes, les nuances de réalisation et les différences de choix en font des systèmes concurrents.

PAROS a été élaboré et réalisé au cours du projet européen **ESPRIT SUPERNODE II (P2528)**, qui a réuni universitaires et industriels. Le prototype qui fut développé au cours de cette collaboration est disponible pour des réseaux de transputers, dont les architectures Supernode font parties. Il constitue le cadre des implémentations et expérimentations effectuées dans cette thèse.

6.1.1 Architecture générale

Sans laisser de côté les autres architectures, la conception de PAROS a pris essentiellement en compte les problèmes que posent les machines parallèles à mémoire distribuée et leur programmation, avec une vision la plus générale possible. Ainsi une machine peut avoir un nombre variable de processeurs, interconnectés par un réseau quelconque et de surcroît *reconfigurable* (où la topologie d'interconnexion est programmable).

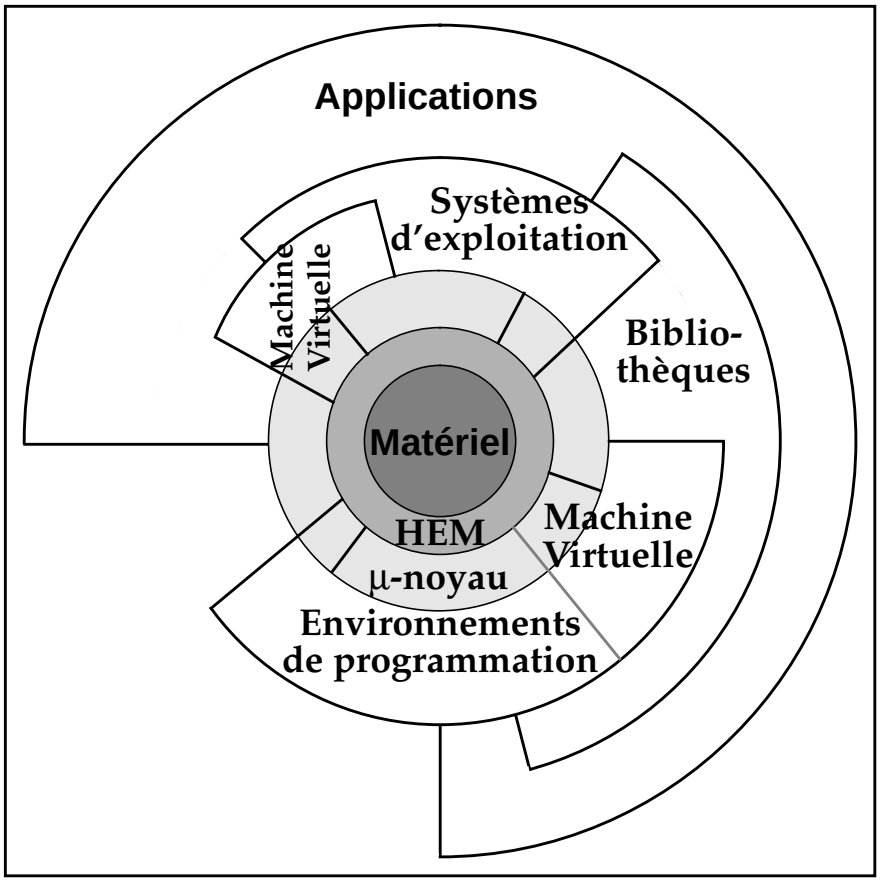


Figure 6.1 : structure générale de PAROS.

Pour proposer une vision rationnelle de ces caractéristiques, sans rendre complexe le développement d'applications, PAROS adopte une architecture structurée et évolutive, illustrée par la figure 6.1. Tout d'abord, afin de rendre la totalité du système aisément portable et apte à fonctionner sur tout type de matériel, la première couche logicielle virtualise une machine parallèle dont les fonctionnalités sont clairement définies et accessibles quel que soit le support d'exécution. Cette couche, dénommée *HEM*[1], donne la vision d'une machine abstraite composée de

1. Hardware Extension Machine (machine d'extension du matériel).

processeurs deux à deux connectés, sans mémoire commune, et disposant du support minimal pour l'échange de messages, ainsi que des fonctionnalités de base pour la gestion de processus (activation, arrêt, ordonnancement primitif, manipulation des contextes d'exécution). Construits au-dessus des services «immuables» du HEM, le reste des composants du système sont de cette manière indépendants du matériel utilisé ; et l'implantation de PAROS sur différentes machines se résume à celle du HEM. En fait, que ce soit la manipulation des processus ou encore la communication, les tâches dévolues au HEM peuvent être directement intégrées dans le matériel. L'expérience des transputers, qui incluent dans un seul circuit une unité de calcul, une mémoire, des ports d'entrée/sortie et un ordonnanceur, nous montre la faisabilité de cette intégration. Mieux, les tous derniers transputers T9000 associés au routeurs C104 offrent déjà le support matériel pour donner aux applications la vision d'une machine totalement connectée.

Dans la démarche suivie le coeur du système est un micro-noyau expurgé des tâches de haut niveau : **ParX**. C'est à ce niveau que sont implémentés le modèle de processus, le modèle de communication, et les politiques d'allocation de ressources (processeurs et mémoire). Nous retrouvons donc le principe des micro-noyaux des systèmes répartis ; cependant, ParX étend ces possibilités pour répondre aux particularités du parallélisme. L'innovation et l'originalité de notre approche repose sur la *généricité* du noyau. En effet ParX se compose d'un micro-noyau fixe, le **π-nucleus**, qui comprend les services de base du modèle de processus et du modèle de communication, ainsi qu'un mécanisme de construction générique de protocoles. C'est à partir de ce mécanisme que sont intégrés dans le noyau, selon les besoins, des services supplémentaires : protocoles de communications, politiques d'ordonnancement, ou encore partage de mémoire. Les services ainsi rajoutés sont fédérés pour former des *machines virtuelles* qui offrent à l'utilisateur plusieurs interfaces et modèles de programmation : mémoire virtuellement partagée, diffusion, client-serveur en sont quelques exemples. La figure 6.2 résume l'organisation de ParX.

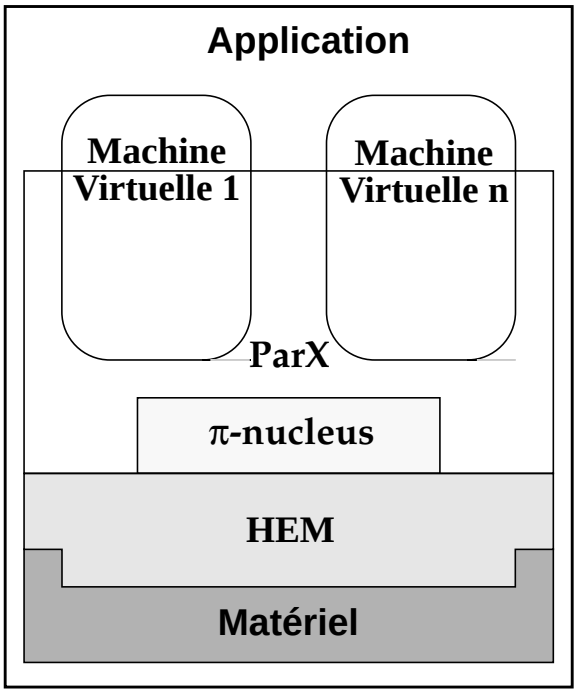


Figure 6.2 : structure du noyau ParX.

Enfin, au-dessus de ParX sont bâtis les outils traditionnels : systèmes d'exploitation, environnements de développement ou bibliothèques. C'est également au-dessus de ParX, que PAROS

gère la machine et son partage entre les utilisateurs. Comme nous le verrons, après avoir présenter les modèles de processus et de communication, les processeurs sont considérés comme des ressources que le système se charge d'allouer aux applications.

6.1.2 Modèle de processus

Le modèle de processus de ParX, comme celui de Peace, comporte trois niveaux; ces abstractions sont le ***thread***, la ***tâche*** (ou task) et la ***Ptâche***[1] (ou Ptask). Il s'agit d'un modèle flexible qui offre les concepts suffisants pour des applications parallèles qui utilisent différents grains de parallélisme.

Flot de contrôle séquentiel, le thread est un processus léger. Du point de vue du noyau, un thread est une unité d'ordonnancement et de pseudo-parallélisme qui s'exécute sur un processeur unique en concurrence avec d'autres threads; les threads forment ainsi un premier niveau d'ordonnancement. Son contexte d'exécution est réduit au minimum et se résume presque essentiellement aux registres du processeur, ce qui rend leur ordonnancement particulièrement simple et rapide, avec des temps de commutation de contextes très faibles. Cette caractéristique permet de surcroît d'employer un ordonnanceur directement intégré dans le matériel.

L'expression du parallélisme se fait à partir des tâches qui regroupent un ensemble de threads dans un même espace d'adressage. La concurrence des threads à l'intérieur d'une tâche, et le contrôle de cette concurrence, fournit un support efficace pour l'implémentation des constructeurs des langages parallèles (le PAR d'Occam par exemple), ou obtenir un recouvrement des calculs et des communications à partir des mécanismes synchrones du noyau. Du fait qu'il sont placés dans un espace d'adressage commun, le partage de mémoire est le principal mécanisme de communication des threads au sein d'une tâche; mais les autres objets de communication leur sont accessibles. Pour les threads qu'elle encapsule, une tâche fournit le véritable contexte d'exécution: c'est à la tâche que sont alloués un certain nombre de ressources (processeur, mémoire, objets de communication, etc.) qui pourront être manipulées par les threads. Pour le système une tâche est aussi une unité de parallélisme et d'ordonnancement; comme les tâches peuvent partager le même processeur physique, elles forment un second niveau d'ordonnancement dont la politique est implémentée dans le noyau et peut être adaptée aux besoins (environnement particulier, temps réel, etc.).

La Ptâche représente une application parallèle en cours d'exécution sur une machine virtuelle; cette machine virtuelle est celle qui est définit par la structure même de l'application: décomposition en tâches et topologie de communication entre ces tâches. Qu'il s'agisse des processeurs, de la configuration du réseau d'interconnexion ou du placement des tâches sur les processeurs, c'est le système qui assure la correspondance entre la machine virtuelle et le support physique réellement alloué à la Ptâche. C'est d'ailleurs là que la généricité du noyau joue son rôle et prend son sens: une Ptâche s'exécute dans une configuration adéquate du micro-noyau, c'est-à-dire avec les machines virtuelles de ParX requises. Elle est donc une entité administrative qui gère les ressources affectées à l'application qu'elle représente, qui contrôle l'exécution de celle-ci depuis le lancement des tâches jusqu'à leur terminaison. De plus elle définit un domaine de communication et de protection, et de ce fait permet d'assurer la sécurité lorsqu'une machine est partagée entre plusieurs applications.

La figure 6.3 illustre le modèle de processus de ParX. Nous avons représenté en pointillés le lien de connexion entre la Ptâche et le reste du système pour dénoter le contrôle auquel il est

1. pour tâche Parallèle.

soumis et qui a pour but de garantir un certain isolement de la Ptâche et la sécurité du système.

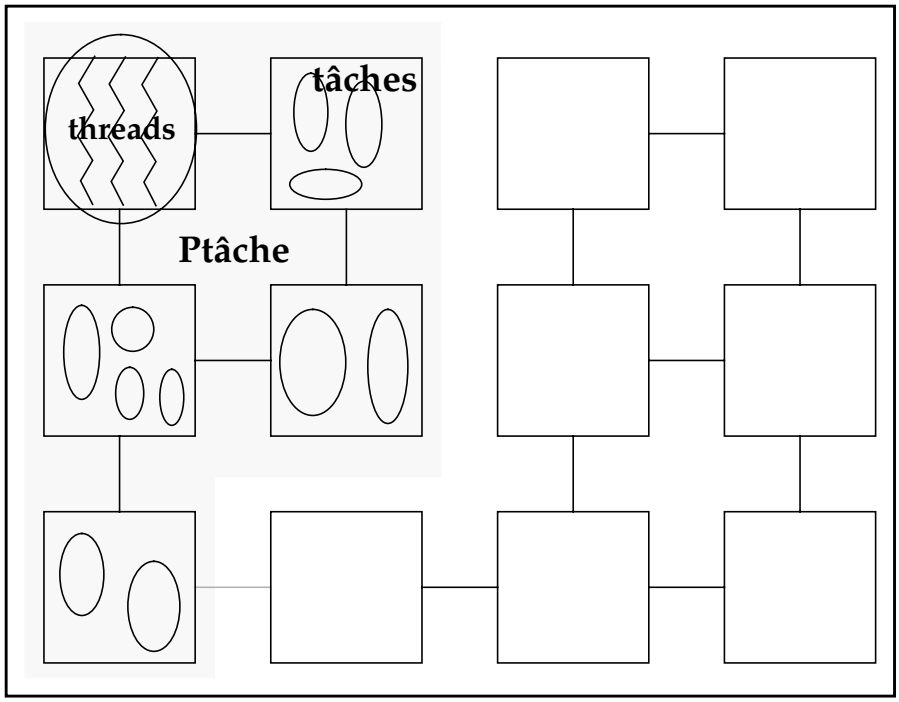


Figure 6.3 : Ptâche, tâches, threads dans ParX.

6.1.3 Modèle de communication

Comme nous l'avons vu le micro-noyau de ParX est générique, et principalement en ce qui concerne les protocoles de communication. Il est donc difficile de parler d'un modèle de communication unique et immuable; le terme «modèle de communication évolutif» correspondrait mieux. Dans ce cadre, les objets de communication disponibles varient selon les machines virtuelles que l'application s'est attachée. Toutefois deux protocoles, présents dans le π-nucleus et donc dans toute instance du noyau, ont été choisis comme objets de communication de base: les ***canaux*** et les ***ports***.

Les premiers implémentent presque à l'identique les canaux d'Occam, soit des échanges de données synchrones entre deux processus. Dans cette réalisation, les canaux sont accessibles quelles que soient les localisations des processus dans le réseau. De plus deux modes sont ici disponibles: unidirectionnel et bidirectionnel.

Egalement synchrones, les ports quant à eux sont d'une utilisation plus flexible et servent à des échanges d'une source quelconque vers une unique destination; il peut servir à des communication de type client-serveur. Bien que les threads d'une tâche puissent y avoir recours pour communiquer, le but principal des canaux est de relier deux tâches entre elles. Ils sont locaux à une Ptâche et ne peuvent en aucun cas être employés comme moyen de communication entre Ptâches; ce qui n'est pas le cas des ports qui sont les seuls objets de communication des Ptâches. La correspondance entre le modèle de processus et ce modèle de communication de base est représentée dans la figure 6.4.

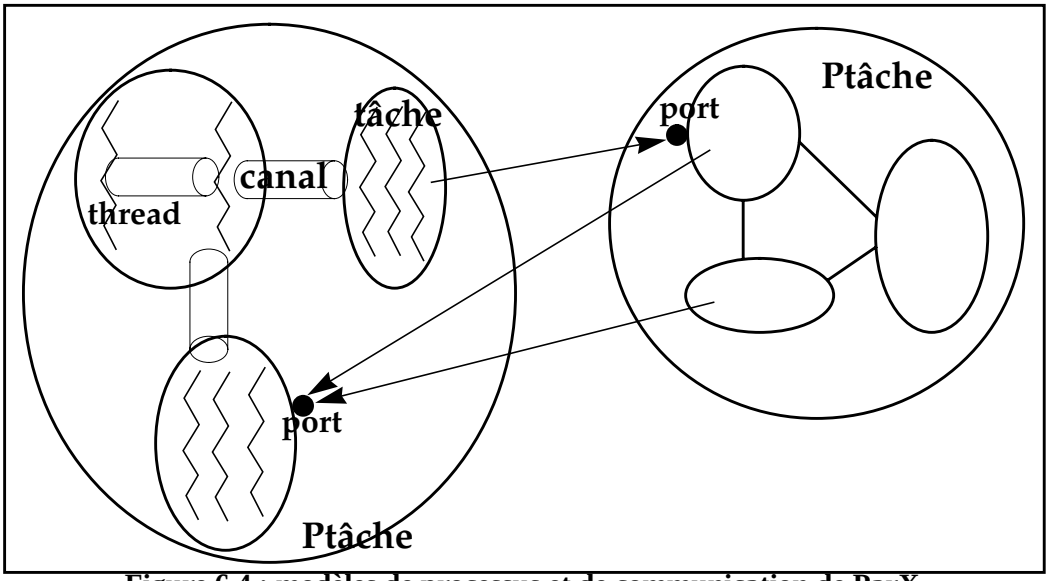


Figure 6.4 : modèles de processus et de communication de ParX.

Parmi les différentes machines virtuelles actuellement disponibles, les protocoles les plus représentatifs sont: le client-serveur (**ASP**[1]), l'accès à une mémoire distante (**DDMA**[2]), et, la **diffusion** qui est le sujet de cette thèse. Construits au-dessus du service de routage et d'acheminement de messages, qui est une fonction du HEM, ils respectent tous, y compris les canaux et les ports, une structure précise.

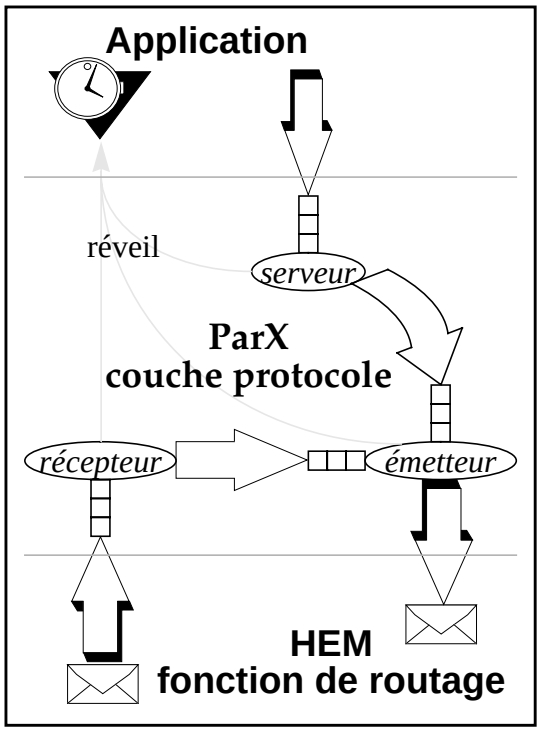


Figure 6.5 : structure générique d'un protocole dans ParX.

Cette structure définit trois processus: un *serveur*, un *récepteur* et un *émetteur*; les processus *récepteur* et *émetteur* relient le protocole à la fonction de routage, alors que le *serveur* se charge des requêtes en provenance de l'application, ou plus exactement de la couche supé-

1. pour **A**synchronous **S**erver **P**rotocol.
2. pour **D**istributed **D**irect **M**emory **A**ccess.

rieure dans l'architecture de PAROS. Elle définit également les files de requêtes entre ces processus, ainsi que la politique de gestion qui leur est associée; c'est-à-dire que pour assurer la correction des protocoles vis-à-vis du routage, pour qu'ils ne puissent pas introduire d'interblocage au niveau de l'acheminement, elle détermine le cadre d'interaction des trois processus (cf. figure 6.5). A partir de ce modèle générique, l'implémentation d'un protocole se fait en décrivant simplement les diverses fonctions de chacun de ces processus.

6.1.4 Gestion des ressources

Pour assurer le partage d'une machine entre plusieurs utilisateurs et plusieurs applications, PAROS considère les processeurs comme des ressources qu'il gère de différentes façons selon l'emploi qui en est fait. Ils sont toujours manipulés en groupe, soit sous forme de *cluster*[1], soit sous forme de *bunch*[2].

Le bunch représente un ensemble de processeurs sans aucun support système; il s'agit en quelque sorte d'une machine nue totalement sous le contrôle de l'utilisateur. Les bunchs servent à exécuter des applications déjà existantes sans avoir à les réécrire, et rendent PAROS compatible avec certains environnements parallèles actuels. Certains utilisateurs «chevronnés» préféreront le bunch pour exécuter leurs programmes avec plus d'efficacité, sans souffrir du surcoût nécessairement introduit par le noyau. Le bunch est donc d'une utilisation très spécifique, et est une «facilité» offerte par le système.

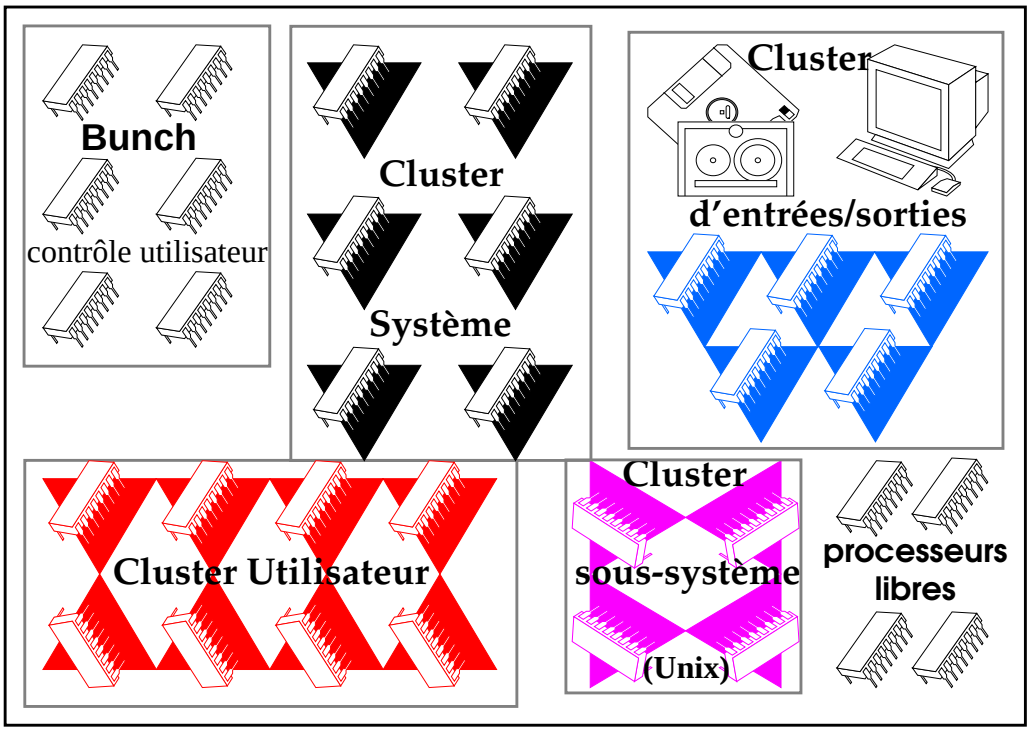


Figure 6.6 : partage d'une machine en Clusters et Bunch avec PAROS.

Pour avoir accès aux services de PAROS, il faut utiliser le cluster. Il regroupe un ensemble de processeurs interconnectés de façon appropriée à l'application qu'il va accueillir, avec le noyau du système dans la configuration requise. Associés, le cluster et la Ptâche forment la machine virtuelle d'exécution d'un programme parallèle. De plus c'est une entité dynamique

1. «grappe».
2. «bouquet».

dans laquelle le nombre de processeurs et leur topologie physique d'interconnexion peuvent varier; du moins lorsque cela est matériellement possible. Cette propriété est intéressante, entre autres pour améliorer l'utilisation des ressources inoccupées, pour augmenter les performances globales du système, ou encore pour supporter des mécanismes de tolérances aux pannes.

La figure 6.6 présente, de manière simplifiée, les possibilités de partage d'une machine qu'offre PAROS. On remarquera en particulier la présence d'un cluster système qui sert d'accès à PAROS et permet aux différents utilisateurs de se connecter, puis la cohabitation de ce cluster avec d'autres sous-systèmes comme Unix par exemple. Il faut noter également l'existence d'un cluster dévolu aux entrées/sorties.

Enfin en ce qui concerne la mémoire, autre ressource importante, ParX ne définit pas de politique de gestion très évoluée, et repousse cette tâche dans la ou les machines virtuelles consacrées à cet effet - une telle machine existe: ***DIVA***; elle offre le partage virtuel de mémoire entre les processeurs [BaMu94]. La mémoire est ainsi simplement organisée en *segments*, qui sont des ensembles d'adresses mémoires contiguës d'un même processeur. Chaque segment possède un identificateur, local à la Ptâche, et peut être cartographié dans l'espace d'adressage des tâches selon une sémantique de **copie**, avec la restriction qu'il ne peut appartenir qu'à un seul espace d'adressage à la fois. L'interface de gestion des segments est accessible aux développeurs système pour, par exemple, effectuer un chargement de code sur les processeurs.

6.2 Interface de PDVM

L'interface de notre support pour la diffusion n'est pas, comme nous l'avons signalé au tout début de ce chapitre, une bibliothèque qui propose une vaste palette d'opérations globales. Elle est simplement constituée d'un ensemble de primitives, aisément manipulables, qui donnent accès aux protocoles de communication de PDVM et offre les fonctionnalités nécessaires à la construction d'autres protocoles ou fonctions.

Cette interface se compose de deux ensembles séparés de primitives écrites en langage ***C***, l'un pour le protocole synchrone et l'autre pour le protocole asynchrone. Alors que l'utilisation du protocole synchrone est totalement indépendante du protocole asynchrone, ce dernier ne peut être dissocié du premier auquel il fait appel. L'association de PDVM à une application peut donc être réduite au seul protocole synchrone, lorsque le second n'est pas requis.

Ceci n'est effectivement possible que parce que la gestion des groupes de processus pour l'un des protocoles est totalement indépendante de celle de l'autre; c'est-à-dire que les groupes de diffusion synchrone sont en tout point distincts des groupes de diffusion asynchrone. Par contre, la sémantique des groupes est identique pour l'un comme pour l'autre.

Du point de vue des interfaces, cela a pour conséquence que les deux bibliothèques proposent des fonctions similaires. Nous commencerons donc par présenter la manipulation des groupes, avant d'introduire les primitives d'initialisation, de communication et de contrôle pour chacun des protocoles.

6.2.1 Groupes de processus

Puisque l'un de nos objectifs est de tendre vers une universalité de la machine virtuelle, la difficulté est de rendre nos groupes souples et adaptables au plus grand nombre d'environnements, dont nous avons pu remarquer les divergences sur ce point (chapitre 2, §2.4). Pour résoudre ce problème, nous n'avons intégré à PDVM qu'un mécanisme de gestion locale des groupes, à partir duquel la construction de véritables structures de groupes, au niveau de l'application, puisse se faire.

Dans cet esprit nous n'avons qu'une interface réduite de primitives de manipulation des groupes. Ces primitives, qui agissent par rapport au processus appelant, ne nécessitent aucune synchronisation entre les processus. Nous obtenons de cette façon une gestion dynamique des groupes dont la composition peut varier au cours de l'exécution et peut recouvrir celles des autres groupes. Ceci rend naturellement possible la construction de groupes comme des listes dynamiques de processus (PVM ou Vartalaap), et permet également de réaliser des groupes ensemblistes (MPI ou CCL) qui requièrent une synchronisation - le protocole de diffusion synchrone s'avère une solution plus générale, aisée et efficace à ce type d'opérations.

Locale à un site, la gestion d'un groupe consiste à maintenir à jour la liste des processus locaux, et uniquement locaux, qui appartiennent au groupe. La composition globale d'un groupe, qui n'est pas directement accessible à l'utilisateur, est donc la concaténation de toutes les listes locales.

Par ailleurs la machine virtuelle garantit la cohérence des communications au sein d'un groupe vis-à-vis de ses membres; c'est-à-dire que la livraison des messages est conditionnée par l'adhésion ou le retrait d'un membre: tout nouveau membre ne sera pas atteint par les messages expédiés avant son adhésion, et un ancien membre ne recevra jamais les messages expédiés après son retrait. De plus, des règles ont été définies pour éviter les interblocages qui pourraient se produire lorsqu'un processus quitte un groupe; elles sont examinées plus en détails ci-après, dans la présentation des primitives de retrait d'un processus.

Dans le prototype de PDVM le nombre de groupes est un paramètre défini comme une constante entière (**`NB_SYNC_BC_GROUP`** et **`NB_ASYN_BC_GROUP`** pour les protocoles synchrone et asynchrone), fixe et donné par l'application lors du chargement de la machine virtuelle. Les groupes sont numérotés par rapport à ce paramètre: de 1 à `NB_SYNC_BC_GROUP` ou `NB_ASYN_BC_GROUP`, et sont identifiés par leur numéro.

En dehors de ces groupes il existe un groupe réservé, de numéro 0, qui comprend tous les processus, et qu'il est impossible de quitter. Il est l'équivalent du groupe **`all`** dans BSP, ou du groupe `MPI_COMM_WORLD` de MPI. Il peut être également très avantageux pour réaliser un lancement synchrone de processus distribués ou implémenter une primitive de terminaison correcte: par exemples la fonction `p4_wait_for_end` de p4, la terminaison synchrone de processus parallèles d'Occam (`PAR`), ou la terminaison correcte des tâches d'une Ptâche de PAROS/ParX.

Seulement deux actions sont accessibles à un processus vis-à-vis d'un groupe: s'y joindre (*Join*) ou s'en retirer (*Leave*). Nous avons donc deux primitives pour chacun des deux protocoles: **`SyncBcast_JoinGroup`** et **`SyncBcast_LeaveGroup`** pour la diffusion synchrone, et, **`AsynBcast_JoinGroup`** et **`AsynBcast_LeaveGroup`** pour la variante asynchrone.

Ces quatre primitives reçoivent comme paramètres l'identification du processus appelant (de type **`Bcast_ident`**) et le numéro de groupe concerné. Elles retournent toutes les quatre une valeur (de type **`t_result`**), qui indique le succès de l'opération ou un code d'erreur. Elles ont le profil suivant:

```
<erreur> = SyncBcast_JoinGroup(<ident>,<groupe>);
<erreur> = SyncBcast_LeaveGroup(<ident>,<groupe>);

<erreur> = AsynBcast_JoinGroup(<ident>,<groupe>);
<erreur> = AsynBcast_LeaveGroup(<ident>,<groupe>);
```

Pour les groupes de diffusion synchrone, l'appel à ces fonctions est soumis à une règle de correction et de cohérence des groupes: *un processus ne peut se joindre ou quitter un groupe alors qu'un message diffusé sur ce groupe est en cours de réception.*

Il n'en est pas de même pour les groupes asynchrones où la règle de correction s'exprime de manière interne: lorsqu'un processus se retire d'un groupe tout se passe comme s'il avait lu tous les messages en attente; ceci dans le but que la libération de l'espace tampon ne soit plus sujette à la réception des messages par ce processus.

6.2.2 Diffusion synchrone

Nous avons vu que les fonctions de manipulation des groupes nécessitent une identification des processus. Il s'agit d'une structure de données, cachée à l'utilisateur, propre à chacun des protocoles. Elle est créée et initialisée pour tout processus lorsqu'il se déclare, normalement au début de son exécution, comme utilisateur du protocole. C'est d'ailleurs à ce moment là qu'il est associé au groupe 0, et qu'il reçoit son identificateur. Dans ParX, cela est réalisé explicitement par l'appel à la primitive **`UseSyncBcast`**:

```
<ident> = UseSyncBcast();
```

Seul le manque d'espace mémoire, pour allouer la structure de données associée à l'identificateur, peut produire une erreur qui est signalée par un identificateur nul (`NULL` en C).

La diffusion d'un message à un groupe se fait en invoquant la primitive **`SyncBcast_send`**, qui est accessible à tout processus, même ceux qui ne sont pas membres du groupe. Elle reçoit en paramètres l'identificateur du processus s'il est membre du groupe ou `NULL` sinon, le numéro du groupe de diffusion, le message ainsi que sa taille (en nombre d'octets). Elle renvoie un code de résultat qui indique soit une erreur, soit le succès de l'émission. Le profil de cette fonction est donc le suivant:

```
<erreur> = SyncBcast_send(  <ident>,<groupe>,
                            <message>,<taille>);
```

Si aucune erreur ne se produit, l'émetteur ne reprend son exécution que lorsque tous les destinataires (tous les membres du groupe, sauf l'expéditeur s'il en est membre) sont au rendez-vous. Le synchronisme de la communication interdit d'avoir plusieurs émetteurs sur un même groupe, situation qui engendrerait un interblocage. Notre protocole détecte ces situations et applique une procédure de correction qui détermine de la manière suivante l'unique émetteur:

- lorsqu'il y a plusieurs tentatives émanant d'un même site, il n'est pris en considération que la première qui lui parvient;

- lorsque les requêtes sont distribuées, celle qui sera prise en compte est élue selon un mécanisme équitable à priorités tournantes.

Pour les autres processus la primitive échoue et renvoie un code de retour qui correspond à cette erreur. Par contre des diffusions simultanées sur des groupes distincts sont permises sans restriction, qu'elles émanent ou non de noeuds différents.

Deux fonctions, accessibles uniquement aux membres du groupe concerné, permettent la réception des messages: **`SyncBcast_receive`** et **`SyncBcast_scatterrecv`**. La première est prévue pour recevoir entièrement un message; la spécification d'une taille plus petite que celle du message diffusé ne provoque pas l'échec de la primitive qui effectue la copie de la taille demandée, mais le reste du message est ignoré pour le processus appelant. De même, lorsque la taille indiquée est plus grande que celle du message, la copie entière du message est réalisée. Cette fonction reçoit en paramètres l'identificateur du processus, le numéro de groupe, la zone de réception du message et sa taille:

```
<erreur> = SyncBcast_receive(   <ident>,<groupe>,
                                <message>,<taille>);
```

La seconde primitive permet de ne recevoir qu'une partie du message, spécifiée par un déplacement (en nombre d'octets) et une taille. Son utilité est principalement de pouvoir opérer un éclatement de données entre les membres d'un groupe comme dans MPI (`MPI_SCATTER`) ou CCL (`scatter`). Cependant il n'y a pas de contrôle associé, c'est-à-dire que certains membres peuvent faire appel à `SyncBcast_scatterrecv` alors que d'autre utilise `SyncBcast_receive`. Son profil d'utilisation est très similaire au précédent:

```
<erreur> = SyncBcast_scatterrecv(   <ident>, <groupe>,
                                    <message>, <taille>,
                                    <déplacement>);
```

Une troisième fonction, **`SyncBcast_testrcv`**, est fournie pour tester si un message est en attente de réception, et effectuer une lecture du début du message; ce qui permet donc de connaître la nature du message en attente. La taille des données qu'il est possible d'obtenir avec cette primitive est limitée et doit être inférieure ou égale à `MAX_PACKET_SIZE / 4 -` où `MAX_PACKET_SIZE` est un paramètre du noyau qui correspond à la taille maximale des paquets que le routeur du HEM accepte -, qui correspond à la fois au premier paquet de données diffusées, et à la taille du tampon de stockage de ce paquet dont chaque groupe dispose en tout noeud. L'avantage de cette fonctionnalité est de rendre plus directement accessible la réalisation de primitives de réception qui opèrent une sélection selon une étiquette. Toutefois la validité de cette information n'est pas totalement garantie; elle peut en effet être remplacée lors de l'élimination des interblocages - le délai induit par cette élimination, quoique court, dépend de la taille de la machine et est donc difficile à borner. Outre cet aspect particulier, l'intérêt majeur de cette primitive est de permettre l'implémentation de commandes gardées. Elle reçoit en paramètres le numéro du groupe, une zone mémoire pour stocker les données à recueillir ainsi que leur taille, et retourne **`SYNC_BC_MSG_TO_RCV`** si un message est attente sur le groupe, **`SYNC_BC_NO_MSG`** sinon, ou bien un code d'erreur. Son profil est le suivant:

```
<résultat> = SyncBcast_testrcv( <ident>, <groupe>,
                                <zone>,<taille>);
```

La dernière primitive de l'interface de la diffusion synchrone sert au processus à se retirer de la liste de ceux qui utilisent le protocole. Normalement invoquée à la fin de son exécution, elle détruit l'identificateur du processus et le supprime du groupe 0. Pour cela il est impératif qu'il se soit retiré de tous les groupes dont il était membre. Cette fonction prend comme unique paramètre l'identificateur du processus:

```
<erreur> = ExitSyncBcast(<ident>);
```

Le tableau 6.1 résume l'ensemble des messages d'erreurs retournés par les fonctions de l'interface synchrone:

code	erreur
SYNC_BC_ERR_NULL_MSG	la réception ou l'émission d'un message de taille nulle n'est pas permise
SYNC_BC_ERR_BAD_GRP	le groupe spécifié n'existe pas, ou il est impossible de quitter le groupe 0
SYNC_BC_ERR_DEAD_LOCK	opération refusée pour garantir l'absence d'interblocage ou la cohérence du groupe
SYNC_BC_ERR_DVM_NOT_USED	mauvais identificateur de processus, le processus n'est pas un utilisateur du protocole synchrone
SYNC_BC_ERR_ALREADY_MEMBER	le processus se joint à un groupe dont il est déjà membre
SYNC_BC_ERR_NOT_MEMBER	le processus ne peut exécuter l'opération que sur un groupe dont il est membre (quitter ou recevoir)

Tableau 6.1 : erreurs de l'interface synchrone de PDVM.

6.2.3 Diffusion asynchrone

Pour les processus utilisateurs de ce protocole, l'obtention d'un identificateur est la première opération à effectuer. De même type `Bcast_Ident` que celui de la diffusion synchrone, cet identificateur a ici une signification bien différente. Il est par conséquent impératif, pour un processus qui utilise les deux protocoles, d'avoir un identificateur pour chacun d'eux. Comme son homologue synchrone la primitive n'attend aucun paramètre, retourne un identificateur nul en cas de d'échec, et intègre le processus appelant au groupe 0:

```
<ident> = UseAsynBcast();
```

Par ailleurs, pour tout noeud qui contient des utilisateurs de la diffusion asynchrone, il est nécessaire d'initialiser le protocole. Cet amorçage, qui doit intervenir avant que les processus locaux ne puissent obtenir leurs identificateurs, n'est en effet pas automatiquement réalisé par le noyau du fait de l'implémentation en mode utilisateur du protocole. Le premier paramètre (`réplication`) de la procédure **`AsynBcast_Start`**, prévue à cet effet, est le nombre de sites sur lesquels le protocole est actif. Les second et troisième définissent les tampons associés à chacun des groupes: `BufSize` est un tableau de (**`NB_ASYN_BC_GROUP`** + 1) éléments où `BufSize[i]` donne la taille (en nombre d'octets) du tampon du groupe `i`, et `Buf` est un tableau de même taille où `Buf[i]` est le tampon du groupe `i`. Mis à part les groupes sans tampon (de taille nulle), les tampons doivent être alloués, dynamiquement ou statiquement, avant

cet appel et ne peuvent pas changer au cours de l'exécution. De plus, il est impératif que tous ces paramètres soient identiques partout où est initialisé le protocole: même nombre de sites actifs, tampons de même taille pour tout groupe. Le profil de cette fonction est le suivant:

```
AsynBcast_Start(<réplication>,<BufSize>,<Buf>);
```

Le protocole de diffusion asynchrone est construit au-dessus du protocole synchrone et utilise trois groupes synchrones: **ASYN_BC_RQST_GROUP** (1), **ASYN_BC_XFER_GROUP** (2) et **ASYN_BC_ACK_GROUP** (3). Ce sont trois groupes réservés qui ne doivent alors en aucun cas être utilisés par d'autres processus pour l'ensemble de la Ptâche.

La primitive de diffusion d'un message, **AsynBcast_send**, est similaire à son homologue synchrone. Elle peut être appelée par tout processus, qu'il soit ou non membre du groupe concerné. Les mêmes paramètres que pour un appel à SyncBcast_send doivent être fournis, seul un booléen supplémentaire (renvoi) spécifie si l'émetteur doit aussi recevoir le message lorsqu'il est membre du groupe - ceci peut éventuellement servir à l'émetteur pour prendre connaissance de la bonne livraison de son message aux sites récepteurs -; c'est alors une obligation pour lui. Un code d'erreur ou de succès est également retourné par cette fonction:

```
<erreur> = AsynBcast_send(  <ident>,<groupe>,
                            <message>,<taille>,<renvoi>);
```

Deux sémantiques d'émission sont associées à cette primitive selon qu'un tampon est ou non alloué au groupe. Lorsqu'un groupe ne dispose pas d'espace tampon l'émetteur reprend son exécution immédiatement après la diffusion de son message; cette opération est totalement non bloquante. Dans le cas ou le groupe est doté d'un tel espace mémoire, l'expéditeur reprend son exécution après que son message ait été recopié dans tous les tampons; l'opération est cette fois bloquante dans le cas où les tampons sont pleins. Comme les tampons sont de taille fixe, il n'est pas permis de diffuser dans un groupe des messages de tailles supérieures à celle des tampons alloués.

Pour la réception de messages, nous retrouvons des primitives semblables à celles du protocole synchrone: la réception simple avec **AsynBcast_receive**, l'éclatement de données avec **AsynBcast_scatterrecv**, et le test de messages en attente avec **AsynBcast_testrcv**. Les paramètres et les fonctions sont rigoureusement identiques à celles de leurs homologues synchrones:

```
<erreur> = AsynBcast_receive(   <ident>, <groupe>,
                                <message>,<taille>);

<erreur> = AsynBcast_scatterrecv(   <ident>,<groupe>,
                                    <message>,<taille>,
                                    <déplacement>);

<erreur> = AsynBcast_testrcv(   <ident>, <groupe>,
                                <zone>, <taille>);
```

Seule la contrainte d'appartenance au groupe est imposée aux processus appelants. Par contre, la sémantique de réception dépend de l'existence d'un tampon pour le groupe concerné. Lorsqu'aucun tampon n'est défini pour le groupe, seuls les récepteurs prêts, i. e. bloqués en attente d'un message, recevront le message diffusé; pour les autres membres il sera définitivement perdu. Dans le cas contraire, la réception concerne le plus ancien message du tampon non

encore lu par le processus. Si le tampon est vide, l'appel est bloquant jusqu'à ce qu'un message soit reçu. De plus, l'ordre de réception des messages dans un groupe est global, c'est-à-dire identique pour l'ensemble des membres du groupe, et quelle que soit leur localisation.

La dernière primitive pour l'interface asynchrone de la machine virtuelle est celle qui met fin à l'utilisation du protocole pour un processus. Comme celle de la diffusion synchrone, elle retire le processus du groupe 0 et impose qu'il ne soit membre d'aucun autre groupe. L'appel à cette fonction se fait de la manière suivante:

```
<erreur> = ExitAsynBcast(<ident>);
```

Le tableau 6.2 donne l'ensemble des messages d'erreurs retournés par ces fonctions:

code	erreur
ASYN_BC_ERR_NULL_MSG	la réception ou l'émission d'un message de taille nulle n'est pas permise
ASYN_BC_ERR_BAD_GRP	le groupe spécifié n'existe pas, ou il est impossible de quitter le groupe 0
ASYN_BC_ERR_DVM_NOT_USED	mauvais identificateur de processus, le processus n'est pas un utilisateur du protocole asynchrone
ASYN_BC_ERR_ALREADY_MEMBER	le processus se joint à un groupe dont il est déjà membre
ASYN_BC_ERR_NOT_MEMBER	le processus ne peut exécuter l'opération que sur un groupe dont il est membre (quitter ou recevoir)
ASYN_BC_ERR_MSG_TOO_BIG	le message est de taille trop grande pour être diffusé, elle dépasse la limite imposée par la taille du tampon
ASYN_BC_ERR_NO_MEMORY	mémoire insuffisante
ASYN_BC_ERR_ALREADY_STARTED	protocole déjà initialisé
ASYN_BC_ERR_MSG_TRUNCATED	le message reçu a été tronqué à la taille spécifiée (erreur non détectée pour les groupes sans tampon)
ASYN_BC_ERR_MSG_TOO_BIG	le message reçu est plus petit que celui attendu (erreur non détectée pour les groupes sans tampon)

Tableau 6.2 : erreurs de l'interface asynchrone de PDVM.

6.3 Implantation de PDVM dans ParX

Après avoir présenté la description de l'interface de notre machine virtuelle à diffusion, nous allons maintenant étudier sa réalisation et son intégration dans le noyau ParX. Nous commencerons par nous intéresser à l'architecture globale de construction des protocoles. Puis nous détaillerons les protocoles de diffusion qui sont utilisés par les primitives de l'interface. Nous

n'avons pas jugé nécessaire de présenter ces primitives puisque leur utilité est simplement d'offrir un accès aisé aux fonctionnalités des protocoles. Afin de faciliter la lecture nous ne donnons ici que l'ébauche des algorithmes mis en oeuvre; les algorithmes complets sont donnés dans l'annexe C. Par ailleurs dans la description des structures données associées aux protocoles nous avons supposé l'existence, dans le langage, du type *liste de données*.

6.3.1 Architecture de PDVM

La structure de notre machine virtuelle est fortement influencée par l'architecture des systèmes distribués, et plus encore par celle de ParX. Outre l'interface qui est une simple bibliothèque de fonctions, elle se compose de deux tâches: l'une pour la diffusion synchrone et l'autre pour la diffusion asynchrone. Seule la première tâche, qui constitue la partie fondamentale de PDVM, est exécutée en mode privilégié et est intégrée génériquement, au-dessus du π-nucleus, dans le noyau de ParX; la seconde s'exécute en mode utilisateur.

Intégré au noyau du système, le protocole synchrone respecte la structure précise que nous avons présentée dans ce chapitre, section 6.1.3. Il comprend donc un processus *serveur* de requêtes, un *récepteur* et un *émetteur* de messages, ainsi que les files d'attentes de requêtes qui leurs sont associées. Il est construit uniquement à partir des fonctionnalités du HEM, à savoir le routage point-à-point et la diffusion entre processeurs. Dans cette structure le contrôle des processus et des files de requêtes est pris en charge par le π-nucleus; il n'est alors nécessaire que de donner les diverses fonctions de chacun des processus. L'interaction entre ces processus, sur un même noeud comme à travers le réseau, s'apparente à un appel de procédure à distance asynchrone. L'invocation de l'une des fonctions de l'un des processus se traduit ici par l'envoi d'un message qui contient la requête, mais n'implique pas l'attente d'une réponse.

Construit à partir du protocole synchrone, le protocole asynchrone est également conçu autour de trois processus qui sont totalement pris en compte par la machine virtuelle. Le second outil de communication utilisé est le port qui fait lui aussi partie du noyau. Le premier processus est un *serveur de requête* qui amorce les diffusions; c'est à ce processus que s'adresse la primitive `AsynBcast_send` pour soumettre une diffusion. Le second est un *serveur d'acquittements* qui s'assure que tous les noeuds peuvent recevoir un message avant que son transfert ne soit entrepris. Le dernier est le *serveur de transfert*, activé par le processus précédent, qui prend en charge une partie de la gestion des tampons et la réception des messages dans ceux-ci; la diffusion effective du message est délégée à la fonction `AsynBcast_send`.

Cette architecture, dont l'approche peut être rapprochée de celle adoptée pour les systèmes distribués dans le projet ESPRIT BROADCAST [ScSa93, ScRi93], est représentée dans la figure 6.7, ci-après. Nous y avons schématisé les interactions entre les processus du protocole asynchrone avec ceux du protocole synchrone et avec les ports de ParX. Sont également représentées les interactions entre le protocole synchrone et le HEM, et la structure de ce dernier.

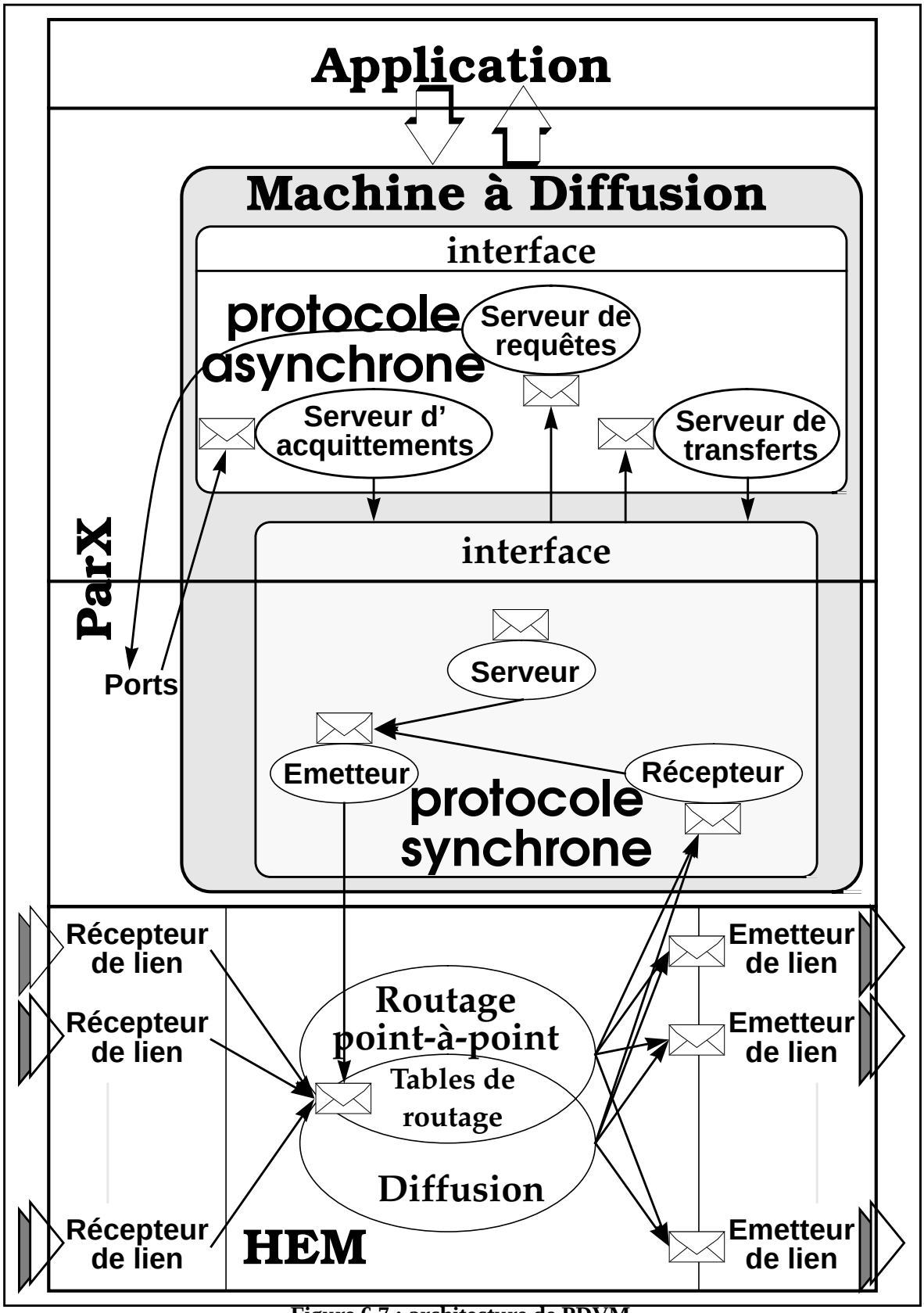


Figure 6.7 : architecture de PDVM.

6.3.2 Le protocole synchrone

Comme nous venons de le voir, seules les fonctions de chacun des trois processus sont nécessaires pour décrire le protocole, nous ne détaillerons par conséquent que ces fonctions. Auparavant nous présentons les structures de données mises en jeu.

Il y a tout d'abord les informations associées à chaque utilisateur du protocole, et initialisées par l'appel à `UseSyncBcast`. Cette structure, de nom **`Membre`**, comprend, hormis les données relatives aux listes de membres gérées pour chaque groupe, la dernière requête **`Req`** du processus.

La structure d'une **`Requête`**, d'émission ou de réception, comprend les données suivantes:

- le code retour de l'opération invoquée: `t_result` **`résultat`**;
- les paramètres de l'opération: `Arguments` **`args`**;
- la structure de données associée au processus appelant: `PROCESS` **`processus`**.

Les **`Arguments`** de toute opération sont les suivants:

- le message de l'utilisateur: `[] BYTE` **`Msg`**;
- la taille du message: `INT` **`Taille`**;
- la spécification de la partie du message qui doit être reçue, exprimée par rapport au début du message: `INT` **`Depl`**;
- un indicateur d'emplacement pour la réception d'un paquet: `INT` **`Début`**;
- le groupe concerné: `INT` **`Group`**;
- l'identification du processus appelant: `Bcast_Ident` **`Ident`**.

Enfin, un **`Groupe`** est constitué des informations ci-dessous:

- la liste des membres locaux du groupe: `Liste de Membre` **`Membres`**;
- les paramètres de la diffusion en cours: `Arguments` **`Param`**;
- la requête de l'émetteur: `Requête` **`ReqEmetteur`**;
- le tampon pour le premier paquet de données envoyé: `[MAX_PACKET_SIZE/4] BYTE` **`tampon`**;
- la taille de la partie du message reçue: `INT` **`TailleReçue`**;
- l'identification du noeud émetteur: `INT` **`ProcEmetteur`**;
- le noeud le plus privilégié dans un conflit entre diffusions concurrentes (cas d'interblocage): `INT` **`PlusPrivilégié`**;
- le nombre de membres locaux: `INT` **`NbMembres`**;
- le nombre de récepteurs locaux prêts à recevoir: `INT` **`NbPrêts`**;
- le nombre d'acquittements reçus: `INT` **`NbAck`**;
- un indicateur d'appartenance au groupe de l'expéditeur (vaut 1 s'il est membre et 0 sinon): `INT` **`EmetteurMembre`**;

- un indicateur de copie du tampon: `BOOL` **`CopieTampon`**;
- un indicateur d'acquittement en cours d'émission: `BOOL` **`Ack`**.

Le *serveur* du protocole se compose de deux fonctions: l'une pour traiter des requêtes de diffusion, et l'autre pour des requêtes de réception. La première consiste à initialiser la communication. Elle place les paramètres de la diffusion dans la structure de groupe, vérifie si l'émetteur est un membre du groupe concerné, contrôle le nombre de récepteurs locaux prêts et active l'expédition du premier paquet de données. Dans cet algorithme, et pour les autres dans ce chapitre, **`n`** est l'identification du processeur local et **`N`** le nombre total de noeuds du réseau.

```
PROC Srv_sync_bc_send(Requête Req)
  SEQ

    -- initialisation des paramètres de la diffusion

    IF
      -- groupe non vide
        SEQ
          -- rechercher l'émetteur parmi les membres
          IF
            -- l'émetteur membre du groupe
              -- le retirer de la liste des membres qui
              -- attendent
          IF
            -- tous les membres locaux sont prêts
              -- acquittement local

      -- groupe vide
        -- acquittement local

    -- diffuser le premier paquet de données
:
```

La deuxième fonction du *serveur* sert à la prise en compte d'un requête de réception. Après avoir contrôlé l'appartenance au groupe du récepteur, elle enregistre un nouveau membre local prêt; puis, si tous les processus locaux du groupe sont prêts active l'acquittement. Dans le cas où cela se passe sur le noeud d'émission, et sous la condition qu'il s'agisse du dernier acquittement attendu, elle peut provoquer le transfert final du message. Le programme est le suivant:

```
PROC Srv_sync_bc_recv(Requête Req)

  IF
    -- le processus appelant n'est pas membre du groupe
      -- retourner une erreur

    -- le processus appelant est un membre du groupe
      SEQ
        -- ajouter un récepteur local prêt
        IF
          -- la diffusion provient de ce site
```

```
            IF
              -- tous les membres locaux sont prêts
                SEQ
                  -- ajouter un acquittement pour ce site
                  IF
                    -- tous les noeuds sont prêts
                      -- envoyer le reste du message

          -- la diffusion ne provient pas de ce site
            IF
              -- il y a une diffusion en attente
              -- et tous les membres locaux sont prêts
                -- envoyer un acquittement
:
```

Le processus *émetteur* quant à lui réalise trois actions différentes: la diffusion du premier paquet et des paramètres de la communication, l'émission d'un acquittement et le transfert final du message. La première fonction, décrite ci-après, est immédiate et consiste simplement à créer un paquet à partir du message et invoquer, par l'intermédiaire de la diffusion du HEM, la primitive de réception correspondante de tous les noeuds. La taille de ce paquet est limitée à celle du tampon contenu dans chaque groupe, c'est-à-dire `MAX_PACKET_SIZE/4` octets.

```
PROC Snd_sync_bc_send_first(Requête Req)
  SEQ
    -- extraire le premier paquet du message
    -- diffuser ce paquet
:
```

La seconde fonction de l'*émetteur* est de réaliser le transfert de tous les autres paquets du message. Lorsque le message est de taille suffisamment petite pour être contenu entièrement dans le premier échange, un paquet de taille nulle est tout de même envoyé pour mettre fin à la communication. Dans le cas contraire, le reste du message est découpé en paquets qui sont envoyés grâce à l'invocation multiple. A la fin du transfert les paramètres du groupe sont réinitialisés pour laisser place à une prochaine communication, et l'émetteur réveillé. Le programme de cette fonction du processus émetteur est:

```
PROC Snd_sync_bc_send_rest(Requête Req)
  SEQ
    -- déterminer la partie du message restant à envoyer
    IF
      -- le message a été complètement envoyé
        -- envoyer un message de fin de diffusion
      -- le message n'a pas été complètement envoyé
        WHILE -- il reste des paquets à envoyer
          SEQ
            -- extraire un paquet non émis
            -- diffuser ce paquet
    -- réveiller le processus émetteur
    -- changer les privilèges affectés aux noeuds
    -- remettre à zéro les paramètres du groupe
:
```

La dernière fonction de ce processus est tout aussi simple que la première et consiste, via le routage point-à-point du HEM, à appeler la primitive de réception des acquittements sur le processeur du processus qui diffuse. Son programme est le suivant:

```
PROC Snd_sync_bc_send_ack(Requête Req)
  SEQ
    -- envoyer au site émetteur, pour le groupe associé
    -- à la requête, un acquittement
:
```

Le processus *récepteur* du protocole possède les fonctions invoquées par l'*émetteur* et le *serveur*, et qui servent à recevoir les différents messages qui lui sont adressés. La première d'entre elles se charge du traitement associé au premier paquet d'un message. Il s'agit d'enregistrer la requête de diffusion et contrôler si tous les récepteurs locaux sont prêts afin d'activer ou non l'acquittement. C'est à ce niveau que sont détectés et résolus les interblocages. Son programme est le suivant:

```
PROC Rcv_sync_bc_recv_first(
    [MAX_PACKET_SIZE/4] BYTE paquet,
    INT noeud,grp,taille)

  IF
    -- nouvelle diffusion
    -- ou plus privilégiée que celle encours
      SEQ

        IF
          -- il y a une diffusion locale en cours
            SEQ
              -- annuler cette diffusion
              -- retourner une erreur à l'émetteur local
              -- remettre à zéro les paramètres du groupe
              -- recalculer les récepteurs locaux prêts

        -- placer le paquet dans le tampon du groupe

        IF
          -- tous les membres locaux sont prêts
            -- envoyer un acquittement au site émetteur

    -- diffusion moins privilégiée que celle en cours
      -- ignorer le paquet
:
```

La réception des paquets suivants par la seconde fonction du récepteur se fait directement dans les espaces mémoires des destinataires. En plus de ce réassemblage des paquets, qui arrivent dans le bon ordre du fait des caractéristiques de notre algorithme de routage (cf. §6.2.2), il effectue la copie du tampon. Le programme de cette fonction est:

```
PROC Rcv_sync_bc_recv_rest([MAX_PACKET_SIZE] BYTE paquet,
    INT taille,grp)

  SEQ
    IF
      -- le tampon du groupe n'a pas été recopié
        SEQ
          -- recopier le tampon dans les espaces mémoire
          -- des récepteurs

    -- placer le paquet reçu dans les espaces mémoire des
    -- récepteurs

    IF
      -- le message a été entièrement reçu
        SEQ
          -- réveiller les récepteurs locaux
          -- changer les privilèges affectés aux noeuds
          -- remettre à zéro les paramètres du groupe
:
```

La dernière fonction de ce processus réceptionne les acquittements, les comptabilise, et lorsque tous les noeuds sont prêts active le transfert final du message:

```
PROC Rcv_sync_bc_recv_ack(INT grp)

  IF
    -- je suis bien l'émetteur de la diffusion à acquitter
      SEQ

        -- ajouter un nouvel acquittement

        IF
          -- tous les noeuds sont prêts
            -- transmettre le reste du message
:
```

Toutes les interactions entre les diverses fonctions des trois processus du protocole définissent un graphe d'états que nous avons schématisé dans la figure 6.8. Nous avons recensé les états actifs: **Diffusion du 1er paquet**, **Diffusion du message** et **Réception du message**, les états inactifs: **Prêt**, **Attente d'un Ack** (acquittement) et **Attente des membres locaux**, ainsi que les différents événements, invocation de l'une des procédures ou conditions, qui régissent le déroulement d'une diffusion et les transitions d'états.

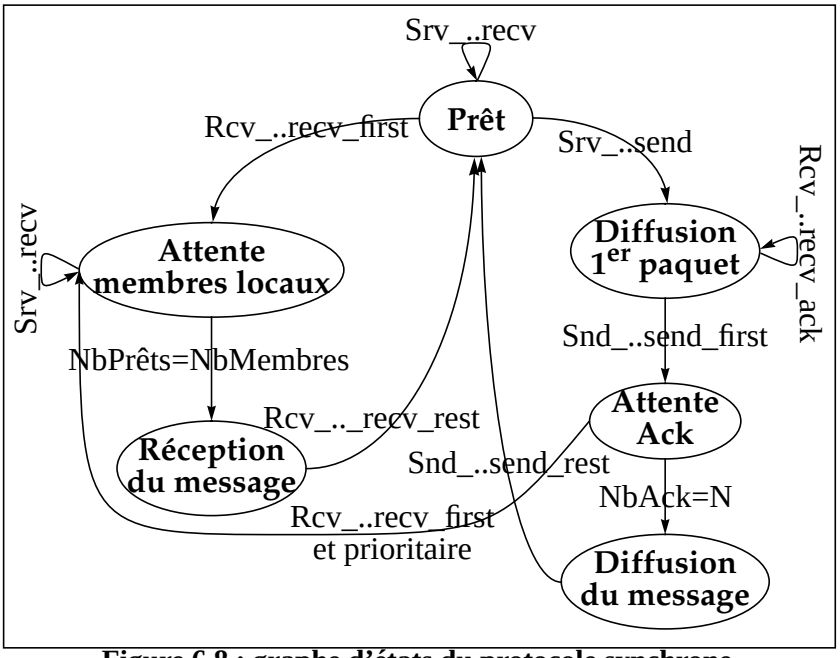


Figure 6.8 : graphe d'états du protocole synchrone.

6.3.3 Le protocole asynchrone

Comme pour son homologue synchrone nous allons tout d'abord détailler les structures de données utilisées par ce protocole. Outre les données employées dans les listes de membres de chaque groupe, à chaque **`Membre`** est associé à un ensemble **`Temps`** d'estampilles temporelles qui marquent, pour chaque groupe, la date de la dernière opération du processus.

Toute **`Requête`** au protocole contient les informations suivantes:

- un indicateur d'appartenance au groupe du processus appelant: `Membre` **`Origine`**;
- le site de la diffusion: `INT` **`ProcEmetteur`**;
- le groupe concerné: `INT` **`Group`**;
- la taille du message: `INT` **`Taille`**;
- le message: `[] BYTE` **`Msg`**;
- un indicateur d'acquittement de la diffusion: `BOOL` **`Ack`**;
- l'identification du processus appelant: `PROCESS` **`processus`**.

Une zone **`Tampon`** à l'intérieur de l'espace alloué à un groupe comprend les données qui suivent:

- l'espace de stockage du message: `[] BYTE` **`zone`**;
- la taille de cet espace: `INT` **`taille`**;
- le nombre de lectures du message attendues: `INT` **`compteur`**;
- la date de création: `INT` **`temps`**;
- la requête de diffusion créatrice: `Requête` **`Req`**.

Un groupe de diffusion asynchrone dispose des informations ci-dessous:

- la liste des membres locaux: `Liste de Membre` **`Membres`**;
- la liste des requêtes de réception en attente de traitement: `Liste de Requête` **`AtteneRcv`**;
- la liste des requêtes d'émission en attente de traitement: `Liste de Requête` **`AttenteSnd`**;
- l'espace tampon de mémorisation des messages: `Liste de Tampon` **`Tampons`**;
- le compteur de temps du groupe: `TIMER` **`horloge`**;
- la taille de tout l'espace tampon: `INT` **`TailleTampon`**;
- le nombre d'acquittements attendus: `INT` **`AttenteAck`**;
- le nombre de membres locaux: `INT` **`NbMembres`**;
- l'état du protocole pour ce groupe: `INT` **`Etat`**;
- l'indicateur de présence d'un tampon associé au groupe: `BOOL` **`SansTampon`**;
- un sémaphore pour la manipulation du groupe: `Sémaphore` **`GroupSem`**;
- un sémaphore pour la manipulation de l'espace tampon: `Sémaphore` **`TamponSem`**;
- la requête de diffusion en cours de traitement: `Requête` **`Req`**.

Le premier processus du protocole est un *serveur de requêtes* de diffusion pour l'ensemble des groupes. Lorsqu'une diffusion est demandée par un appel à `AsynBcast_send`, tous les serveurs du réseau reçoivent, par l'intermédiaire du protocole synchrone, la requête. Si une diffusion est déjà en cours sur le groupe, le site d'émission place la requête dans une file d'attente alors que les autres l'ignorent purement. Dans le cas contraire le traitement est dépendant de la présence d'un tampon et de l'espace disponible dans celui-ci : si aucun tampon n'est alloué au groupe ou si la place restante dans le tampon est suffisante pour y stocker le message, un acquittement est renvoyé au site émetteur, sinon la diffusion est mise en attente d'une libération suffisante du tampon.

```
PROC Serveur_Requête(Bcast_Ident id)

  WHILE TRUE
    SEQ
      -- recevoir une requête de diffusion sur un groupe
      -- entrer en section critique sur TamponSem du groupe

      IF
        -- pas d'autre diffusion en cours sur le groupe
          SEQ
            -- passer à l'état RECEPTION EN COURS
            IF
              -- le groupe n'a pas de tampon
              -- ou il y a assez d'espace
                -- envoyer un acquittement à l'émetteur
```

```
                -- pas assez d'espace dans le tampon
                  passer à l'état ATTENTE DU TAMPON

            -- il y a déjà une diffusion en cours
              IF
                -- cette requête a été émise localement
                  -- différer cette requête en la plaçant
                  -- dans la file d'attente AttenteSnd du groupe

          -- sortir de la section critique
    :
```

Le *serveur d'acquittements* se charge de comptabiliser les noeuds prêts à recevoir le message, et lorsqu'ils sont tous prêts d'activer le transfert sur tous les sites:

```
    PROC Serveur_Ack

      WHILE TRUE
        SEQ
          -- recevoir un acquittement sur un groupe
          -- retrancher cet acquittement de ceux attendus

          IF
            -- plus aucun acquittement attendus sur le groupe
              SEQ
                -- retourner l'acquittement à l'émetteur
                -- réveiller l'émetteur
                -- diffuser un message de contrôle pour activer
                -- le transfert
                -- remettre à jour le nombres d'acquittements
                -- à attendre
    :
```

Pour les groupes sans tampon le *serveur de transfert* réceptionne le message diffusé et ne le transmet qu'aux destinataires locaux en attente de réception; si aucun des membres locaux n'est prêt le message est éliminé. Dans tous les autres groupes, la réception se fait dans une zone mémoire allouée dans le tampon et correctement initialisée; les récepteurs locaux en attente sont alors débloqués.

```
PROC Serveur_Transfert(Bcast_Ident id)

  WHILE TRUE
    SEQ
      -- recevoir le message d'activation du transfert
      -- entrer en section critique sur TamponSem
      -- passer à l'état RECEPTION

      IF
        -- le groupe n'a pas de tampon
          IF
            -- aucun récepteur prêt à recevoir
              -- éliminer le message
            -- il y a au moins un récepteur prêt
              SEQ
                -- recevoir le message dans l'espace
                -- mémoire du premier récepteur prêt
                -- le transmettre aux autres destinataires
                -- réveiller tous les récepteurs locaux

        -- groupe avec tampon
          IF
            -- aucun membre local n'est abonné au groupe
              -- éliminer le message

            -- il y a des membres locaux du groupe
              SEQ
                -- allocation d'une zone dans le tampon
                -- réception du message dans cette zone
                -- estampillage de la zone
                -- réveil des récepteurs en attente

      IF
        -- une diffusion locale est attente pour le groupe
          SEQ
            -- réveil de la première diffusion en attente

      -- revenir à l'état PRET
      -- sortir de la section critique sur TamponSem
:
```

Puisqu'il y a deux modes d'utilisation de ce protocole selon qu'un groupe dispose ou non d'un tampon, nous avons établi deux graphes d'états (voir figures 6.9 et 6.10). Le premier correspond aux groupes sans tampon ou se dégage cinq états : **Prêt**, **Envoi d'un Ack**, **Attente des Ack**, **Diffusion du signal de transfert** et **Réception du message**, qui sont contrôlés par les événements suivants: réception d'une **requête**, réception d'un **signal de transfert**, réception d'un **Ack**, ainsi que par les conditions de transitions: **tous les Ack**[1] **reçus**, **requête locale** ou **requête distante** et **message déposé**. Par ailleurs nous remarquerons que la diffusion du message n'y figure pas, celle-ci étant effectuée en dehors du protocole par la primitive d'émission de l'interface.

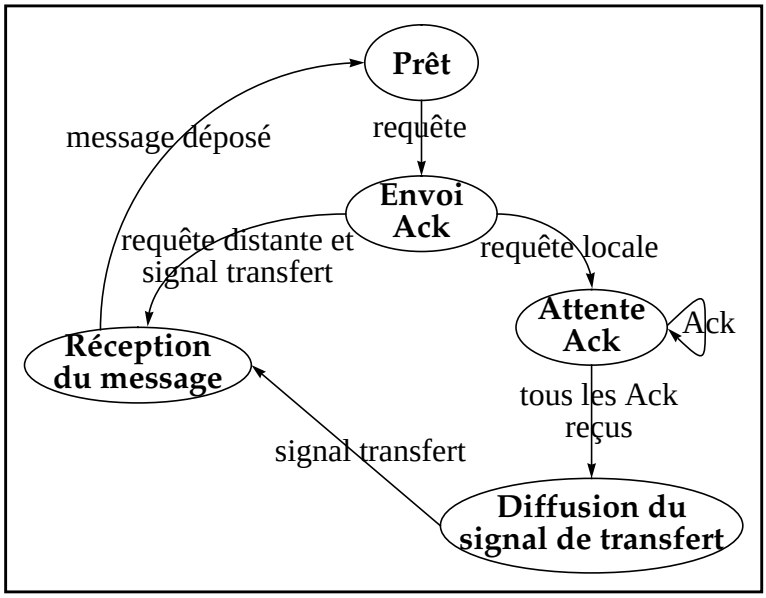


Figure 6.9 : graphe d'états du protocole asynchrone sans tampon.

Pour les groupes avec tampon (figure 6.10), nous obtenons les mêmes états, avec un état supplémentaire pour prendre en compte l'attente de libération des tampons: **Attente tampon**. De même, certaines transitions changent:

- la réception d'une requête s'accompagne des conditions sur l'espace **tampon**: **suffisant** ou **insuffisant**;
- le retour à l'état prêt ne se fait plus lorsque le message a été déposé, mais lorsqu'il a été copié dans le tampon (**message copié**).

1. acquittements.

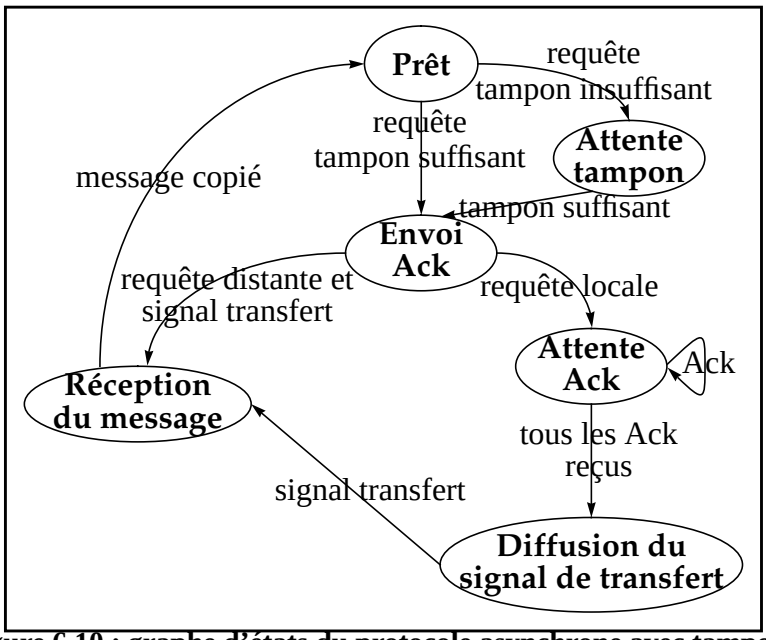


Figure 6.10 : graphe d'états du protocole asynchrone avec tampons.

6.4 Conclusion

Ce chapitre propose une architecture et une réalisation de machine virtuelle adaptée aux communications par diffusion et, plus largement, aux communications globales. Il montre sa faisabilité et son implantation dans le système parallèle PAROS/ParX. Sa généralité en fait le support de construction de multiples interfaces et de multiples systèmes à diffusion tels que ceux que nous avons présentés dans les chapitres 1 et 2.

Dans cette architecture nous retiendrons en premier lieu la modularité obtenue à partir de la décomposition: routage à diffusion, protocole synchrone et finalement protocole asynchrone. Cela permet, selon le niveau, de s'abstraire de contraintes difficiles à gérer : l'acheminement pour la diffusion synchrone ou l'ordre global pour la variante asynchrone. Nous remarquerons que dans la structure de ParX, nos protocoles restent corrects: ils ne génèrent pas d'interblocages et conservent la cohérence des groupes vis-à-vis des communications - tous les destinataires d'un message sont atteints et il n'y a pas de récepteurs indésirables. Nous noterons également une gestion souple des groupes de processus qui rend possible divers styles d'interfaces.

Nous pouvons de plus concevoir une amélioration de nos protocoles au sein d'une gestion de groupes plus étoffée, si nous supposons l'emploi de l'un des routeurs à diffusion basés sur les intervalles que nous avons proposés au chapitre 6. En effet, avec un routeur de ce type nous pouvons effectuer une diffusion sélective plutôt qu'une diffusion à tous les processeurs. Il suffit pour cela de conserver, pour chaque groupe de processus, un intervalle qui définit l'ensemble des processeurs utilisés par les membres du groupe, et de n'effectuer la diffusion des messages que sur cet intervalle.

Evidemment il est loin d'être certain que la majorité des groupes soient répartis efficacement au point de minimiser l'étendue des intervalles, et par voie de conséquence les communications. Nous pensons toutefois, au regard de l'importance croissante de cette catégorie de communications, que l'introduction de ce facteur dans les algorithmes de placement - le lecteur peut se reporter à [Talbi93] pour une étude sur les algorithmes de placement - pourrait alors rendre particulièrement efficace une telle amélioration.

Chapitre 7 : Analyse des Performances de la Machine Virtuelle

Ce dernier chapitre, consacré à PDVM, va nous permettre d'apprécier les performances réelles de la machine virtuelle. A partir de l'ensemble de nos mesures nous allons déduire le comportement de la machine virtuelle dans plusieurs situations. Toutefois, nous ne nous intéressons pas à l'ensemble de ses composants, mais simplement au plus représentatif d'entre eux: le protocole synchrone.

Le choix de ce protocole s'explique en premier lieu par sa position intermédiaire entre le routeur à diffusion et le protocole asynchrone, position qui va nous permettre d'obtenir une approximation des performances du routeur. Certes il ne s'agira que d'une approximation, mais des mesures plus exactes n'apporteraient que peu de précisions supplémentaires. Elles n'auraient de toute manière qu'une importance relative, puisqu'elles ne correspondraient qu'à l'implantation logicielle[1] du routeur dont l'objectif est d'être intégré dans un circuit afin d'obtenir de bien meilleurs résultats. De même l'évaluation du protocole asynchrone ne nous semble pas indispensable; son intérêt est en effet de permettre un recouvrement des calculs et des communications, ce qui est ici réalisé, du fait de notre choix d'implémentation[2], aux dépends des temps de latence d'expédition et de réception des messages.

Pour obtenir nos évaluations nous avons utilisé une machine Supernode de type T-Node avec 18 transputers T800 à topologie d'interconnexion reconfigurable. Chacun des T800, crédités d'une puissance théorique sur 64 bits de 1,1 Mflops, possède une horloge à 20 Mhertz, quatre liens de communication bidirectionnels avec un débit de 10 Mbits/s, une unité intégrée de calcul en virgule flottante sur 64 bits. Tous nos programmes de mesures ont été développés au-dessus du micro-noyau ParX.

Nous mesurerons pour commencer le débit de la diffusion synchrone par rapport à différentes tailles de messages. Ensuite, dans les mêmes conditions, nous étudierons les différences de débits obtenues selon le processeur, c'est-à-dire selon la situation géographique dans un réseau. Puis nous observerons, dans une topologie en grille torique, les répercussions du nombre de processeurs dans le réseau. Nous enchaînerons par une comparaison des performances du protocole dans différentes topologies d'interconnexion. Nous aborderons alors l'influence de la fonction de routage point-à-point sur le temps de diffusion. Après quoi nous comparerons les résultats de notre diffusion synchrone avec ceux du protocole Occam de ParX[3]. Nous ter-

1. implantation dans laquelle la couche de routage est non seulement basée sur un acheminement de type store-and-forward, mais également exécutée en concurrence avec l'application.
2. implémentation au-dessus du protocole synchrone, qui permet de garantir aisément la correction du protocole.
3. protocole de communication point-à-point également synchrone (rendez-vous), dont ParX propose une implémentation distribuée.

minerons nos mesures par l'étude du comportement du protocole dans le cas de diffusions concurrentes. Mais avant cela, nous allons présenter les algorithmes employés pour obtenir toutes les mesures.

7.1 Programmes de mesure

Notre premier programme, qui sert de base à tous les autres, comprend deux processus: l'un qui effectue les diffusions, et le second qui les réceptionne. Le processus émetteur est exécuté par l'un des noeuds du réseau et consiste, après une phase de synchronisation, à diffuser un nombre **X** de fois un même message. Il mesure ainsi le temps moyen de diffusion à partir d'un processeur donné et pour une taille de message déterminée.

```
PROC Diffuseur(Bcast_Ident id,[] BYTE msg,INT taille,grp)
  INT tempsDépart, tempsArrivée, i:
  REAL tempsTotal
  TIMER horloge:
  SEQ
    -- synchronisation
    SyncBcast_send(id,grp,msg,taille)

    -- répétition de la diffusion
    horloge?tempsDépart
    SEQ i = 0 FOR X
      SyncBcast_send(id,grp,msg,taille)
    horloge?tempsArrivée

    -- calcul du temps de communication
    tempsTotal := (tempsArrivée MINUS tempsDépart)
:
```

Le second processus, répliqué sur l'ensemble des processeurs, se synchronise avec l'émetteur et enchaîne les réceptions du même message:

```
PROC Récepteur(Bcast_Ident id, [] BYTE msg, INT taille,grp)
  INT i:
  SEQ
    -- synchronisation
    SyncBcast_receive(id,grp,msg,taille)

  -- répétition des réceptions
  SEQ i = 0 FOR X
    SyncBcast_receive(id,grp,msg,taille)
:
```

La majeure partie de nos mesures a été effectuée avec ce programme. Nous l'avons modifié pour qu'il puisse réaliser des diffusions simultanées en multipliant les processus: l'émetteur est alors dupliqué, sur tous les sites sources, et les récepteurs sont répliqués sur tous les noeuds autant de fois qu'il y a d'émetteurs (un pour chaque groupe).

Pour obtenir des mesures comparables entre la diffusion et les canaux point-à-point synchrones d'Occam, plus exactement de leur implémentation distribuée dans ParX, nous avons utilisé des

algorithmes similaires dans ce dernier programme. Nous retrouvons alors un processus émetteur qui calcule une moyenne des échanges point-à-point sur l'ensemble des noeuds, et un récepteur pour chacun des **N** noeuds.

```
PROC Emetteur([N] CHAN OF ANY c,[] BYTE msg,INT taille)
  INT tempsDépart, tempsArrivée,i , j:
  REAL tempsTotal:
  TIMER horloge:
  SEQ
    -- synchronisation
    SEQ i = 0 FOR N
      c[i]!msg

    -- enchaînement des communications, noeud par noeud
    horloge?tempsDépart
    SEQ i = 0 FOR N
      SEQ j = 0 FOR X
        c[i]!msg
    horloge?tempsArrivée

    -- valcul du temps total
    tempsTotal := tempsArrivée MINUS tempsDépart
:

PROC Récepteur(CHAN OF ANY c, [] BYTE msg, INT taille)
  INT i:
  SEQ
    -- synchronisation
    c?msg

    -- enchaînement des réceptions
    SEQ i = 0 FOR X
      c?msg
:
```

7.2 Débit en fonction de la taille du message

Pour cette première étude nous avons utilisé notre algorithme de mesure pour des tailles de messages comprises entre 8 octets et 64 kilo-octets (ko). Afin que notre programme puisse calculer des temps moyens fiables, le facteur de répétition (X) a été fixé au seuil de stabilité des mesures, soit 2500 messages. Pour chaque taille T de messages le débit moyen D est obtenu en cumulant les mesures des temps de diffusion t_i (exprimés en secondes) sur l'ensemble des N noeuds du réseau :

$$D = \frac{N \times X \times T}{\sum_{i=0}^{N-1} t_i}$$

La topologie d'interconnexion employée est un tore 4x4 auquel sont ajoutés deux processeurs supplémentaires pour effectuer la connexion à une machine hôte et le contrôle du réseau (cf. figure 7.1).

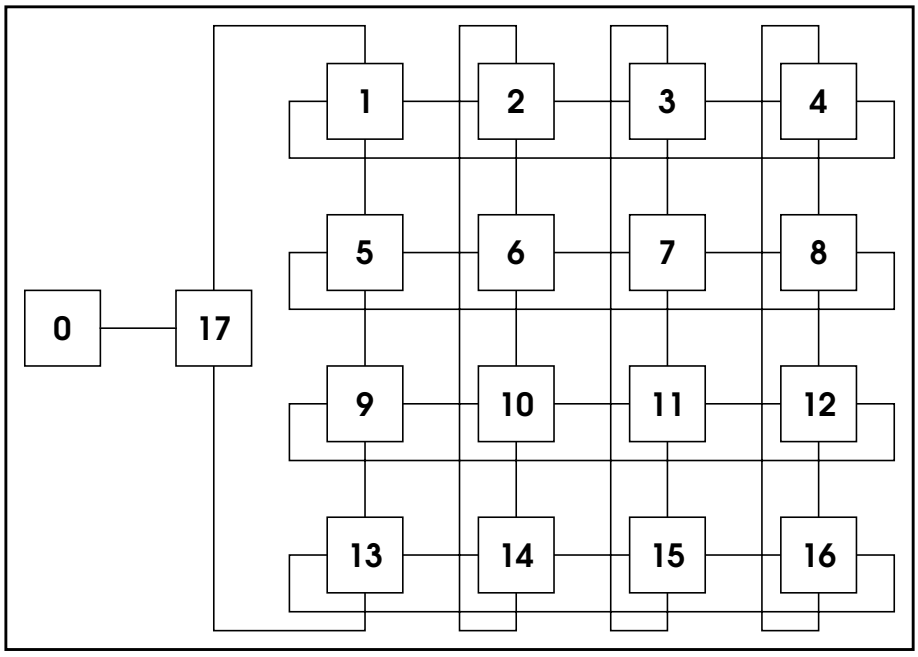


Figure 7.1 : tore 4x4 modifié.

Pour le routeur à diffusion nous avons considéré la fonction de routage point-à-point de l'arbre recouvrant (cf. chapitre 3, §3.3). Avec ces paramètres nous avons pu déterminer les débits moyens pour chaque taille de messages; ils sont reportés dans le tableau 7.1 et sur la courbe de la figure 7.2. Nous remarquons tout d'abord que pour des tailles réduites, inférieures à 512 octets, le débit reste faible; ceci est la conséquence du coût de la synchronisation, en grande partie dû aux acquittements. Pour des tailles moyennes, entre 512 octets et 8 ko, le débit augmente rapidement jusqu'à dépasser 150 ko/s, ce qui correspond à environ le dixième de la bande passante maximum des liens de communications des transputers: 10 Mbit/s, soit 1280 ko/s. Cette évolution rapide est le fait de l'amortissement du coût de la synchronisation.

Avec des tailles supérieures, l'amélioration se fait moindre pour tendre vers 200 ko/s, soit le cinquième du débit maximum des liens de communications. Cette limite s'explique par l'utilisation intensive du réseau pour le transfert du grand nombre de paquets des messages, c'est-à-dire la surcharge des routeurs.

Taille	8 octets	512 octets	1 ko	2 ko	4 ko	8 ko	16 ko	32 ko	64 ko
Débit (ko/s)	1.96404	64.6552	80.7899	107.271	139.535	164.009	179.686	188.753	193.637

Tableau 7.1 : débit en fonction de la taille du message.

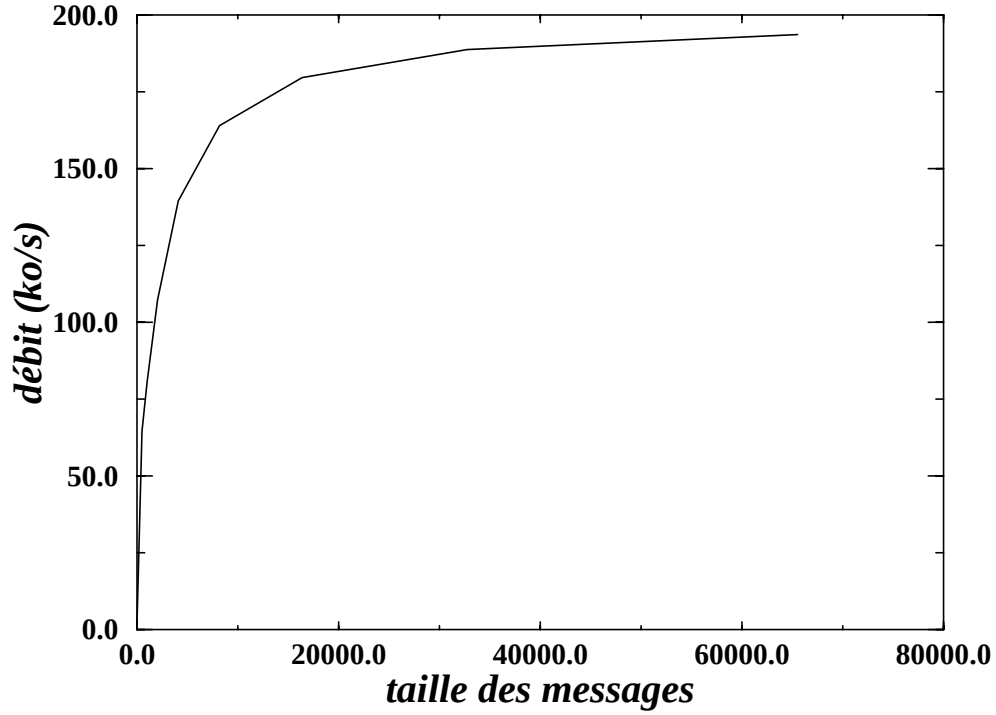


Figure 7.2 : débit en fonction de la taille du message.

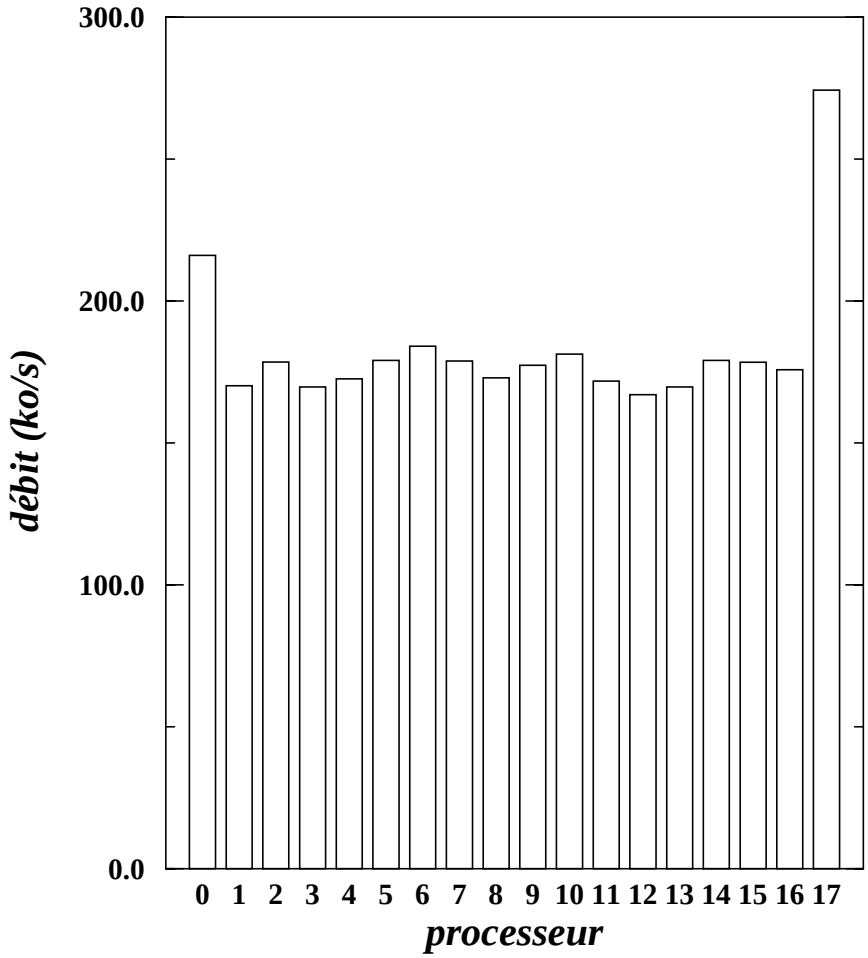


Figure 7.3 : répartition du débit selon le noeud.

7.3 Répartition du débit selon le noeud du réseau

Dans les mêmes conditions que précédemment: même réseau de 18 processeurs, 2500 messages par mesure, tailles de messages comprises entre 8 octets et 64 ko, etc., nous avons calculé pour chaque processeur *i* le débit moyen d_i à partir des temps total t_m de diffusion des X messages de taille T_m:

$$d_i = \frac{X \times \sum T_m}{\sum t_m}$$

Les résultats sont reportés dans l'histogramme de la figure 7.3. Nous remarquons que seuls les processeurs 0 et 17 bénéficient d'un meilleur débit, qui s'explique par la situation géographique de ces noeuds dans le réseau: ils sont en effet situés à la «racine» du tore.

Pour les autres sites les débits varient très peu: de 167 ko/s à 185 ko/s. Notre algorithme, en association avec la fonction de routage point-à-point, garde donc une distribution équitable des noeuds dans cette topologie régulière.

7.4 Influence du nombre de noeuds

Pour étudier les répercutions du nombre de noeuds sur les performances de notre protocole de diffusion, nous avons effectué les mêmes mesures qu'à la section 7.2 sur une même topologie d'interconnexion, la grille torique, en faisant varier le nombre de processeurs. Le T-node utilisé comprenant au maximum 18 transputers, nous avons été réduit à évaluer l'évolution du débit pour 6 processeurs avec un tore 2x2 (cf. figure 7.4), 11 avec un tore 3x3 (cf. figure 7.5) et 18 avec le tore 4x4 (cf. figure 7.1).

La figure 7.6 représente les courbes calculées pour ces trois cas. Nous noterons qu'entre 11 et 18 noeuds la différence de débits est très faible, et pour 6 noeud les débits sont légèrement meilleurs. Nous pouvons déduire de cette expérience que le nombre de noeuds n'est pas ici un facteur important de dégradation des performances. En effet, l'amélioration notée avec 6 processeurs est essentiellement due au fait qu'entre certains processeurs il existe deux liens de communication au lieu d'un seul, ce qui permet de mieux distribuer la charge des communications générées.

Pour généraliser cette conclusion et avoir plus d'éléments d'appréciation, nous pourrions multiplier d'avantage le nombre de noeuds avec des architectures Supernodes comprenant plusieurs tandems. Nous pensons que dans les conditions de nos mesures, et en fonction de que nous venons d'observer, nous noterions une dégradation des performances peu rapide, qui serait la conséquence de la multiplication des délais de traversé des processeurs.

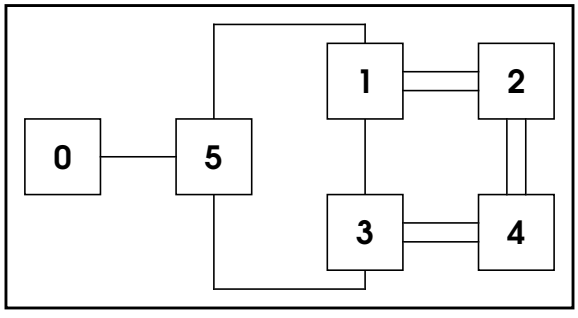


Figure 7.4 : tore 2x2 modifié.

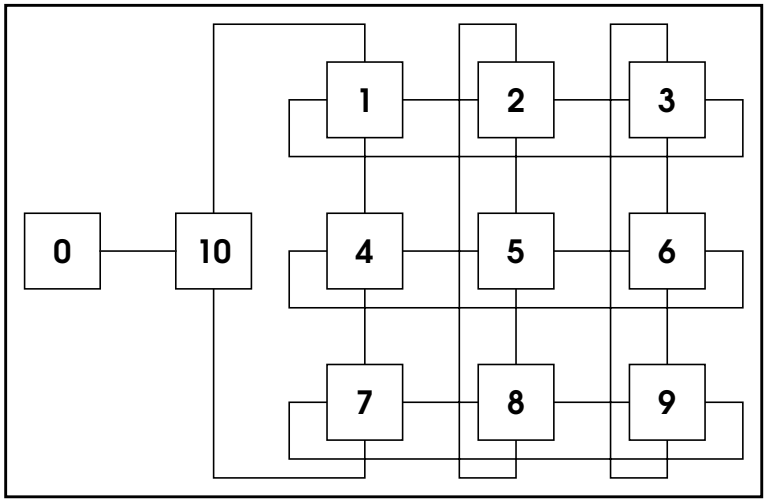


Figure 7.5 : tore 3x3 modifié.

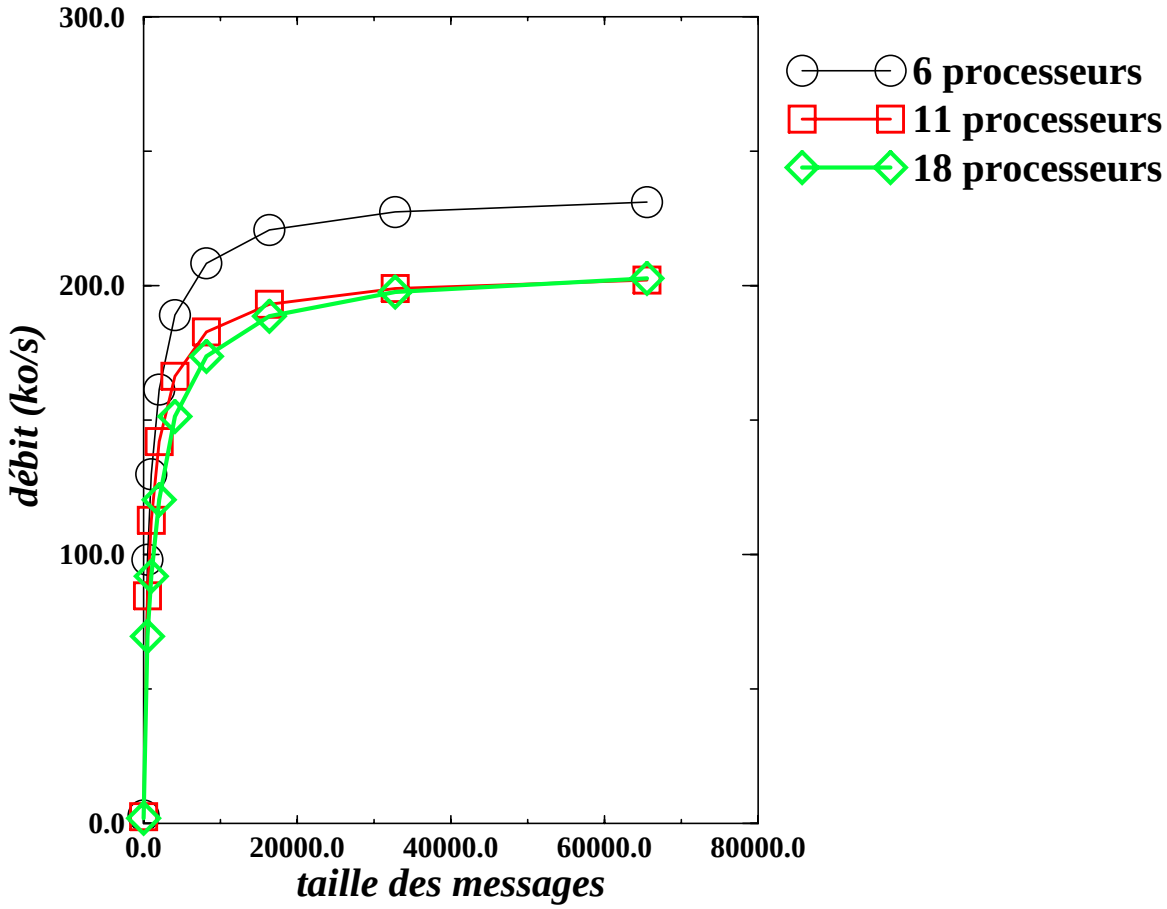


Figure 7.6 : influence du nombre de noeuds.

7.5 Impact de la topologie d'interconnexion

L'importance de la structure du réseau vis-à-vis d'un placement donné des processus sur les communications point-à-point est évidente: plus le réseau d'interconnexion est proche du graphe de communication, plus les processus qui échangent des messages sont dans un voisinage proche, et meilleures sont les performances globales de l'application. Par contre si nous occultons le placement des processus dans un groupe de diffusion et que nous considérons un éclatement du groupe sur l'ensemble du réseau, l'influence de la topologie d'interconnexion est moins claire. Pour l'étudier nous avons comparé les performances de la diffusion synchrone avec différentes topologies.

Trois réseaux, qui comportent le même nombre de processeurs, ont été considérés ici: le tore 4x4 de la figure 7.1, la chaîne de 18 noeuds représentée dans la figure 7.7, et un troisième réseau, schématisé dans la figure 7.8, que nous avons surnommé «losange» en raison de sa structure particulière.

Le critère de comparaison retenu est le débit moyen suivant la taille des messages, mesuré dans des conditions similaires à celles de la section 7.2. Les résultats obtenus sont ceux de la figure 7.9. Nous remarquons une très faible différence entre le tore et le losange, alors que très rapidement la chaîne exhibe des performances nettement supérieures: de l'ordre du double. Pour ce dernier cas, l'explication est identique à celle de l'amélioration que nous avons notée pour le tore 2x2: la présence de deux liens de communications entre les noeuds permet de mieux distribuer les communications entre les liens, et ainsi de réduire l'effet de saturation induite par la diffusion.

Le caractère adaptatif de notre algorithme de diffusion vis-à-vis de la charge du réseau est donc ici très clairement mis en évidence: lorsque les messages comportent plusieurs paquets, ceux-ci transitent en tout noeud par le lien libre parmi les deux possibles; l'acheminement de tout le message se fait alors plus efficacement. Ceci nous montre d'ailleurs les possibilités d'amélioration des performances que nous pourrions obtenir avec un circuit routeur séparé.

Ce qui différencie les réseaux d'interconnexion à l'égard de la diffusion semble donc être lié plus au degré des noeuds et à la duplication des liens entre eux, c'est-à-dire à leur capacité d'exploitation d'un routage adaptatif, qu'à leur topologie d'interconnexion.

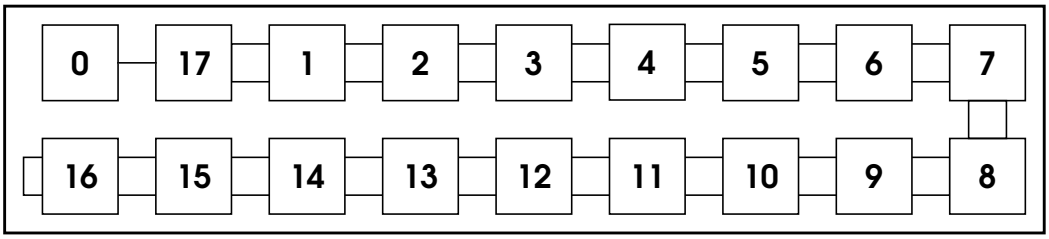


Figure 7.7 : chaîne de 18 processeurs.

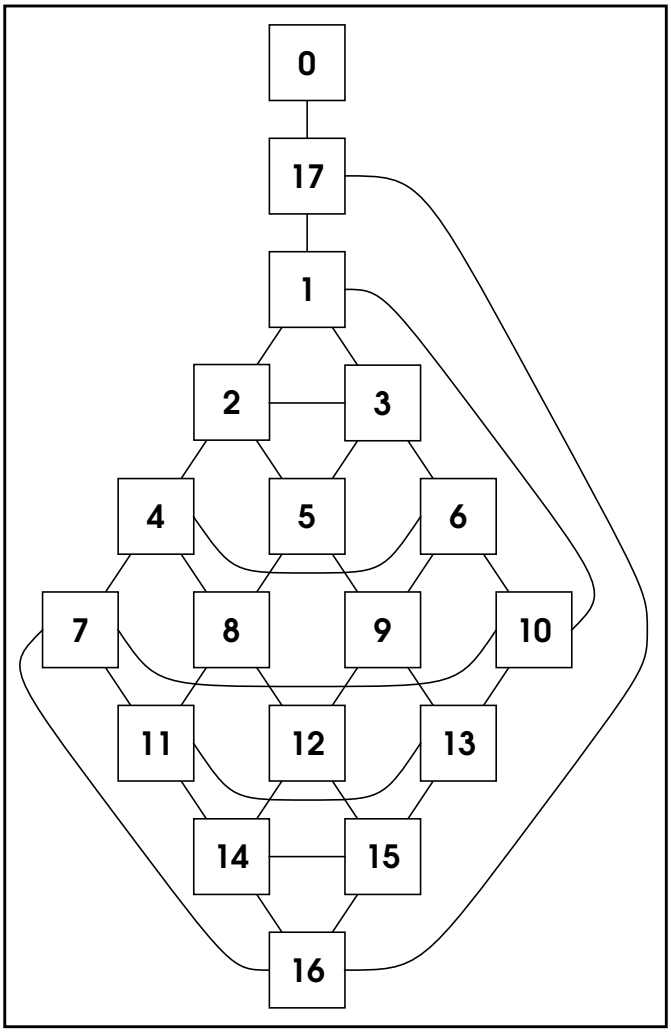


Figure 7.8 : losange de 18 processeurs.

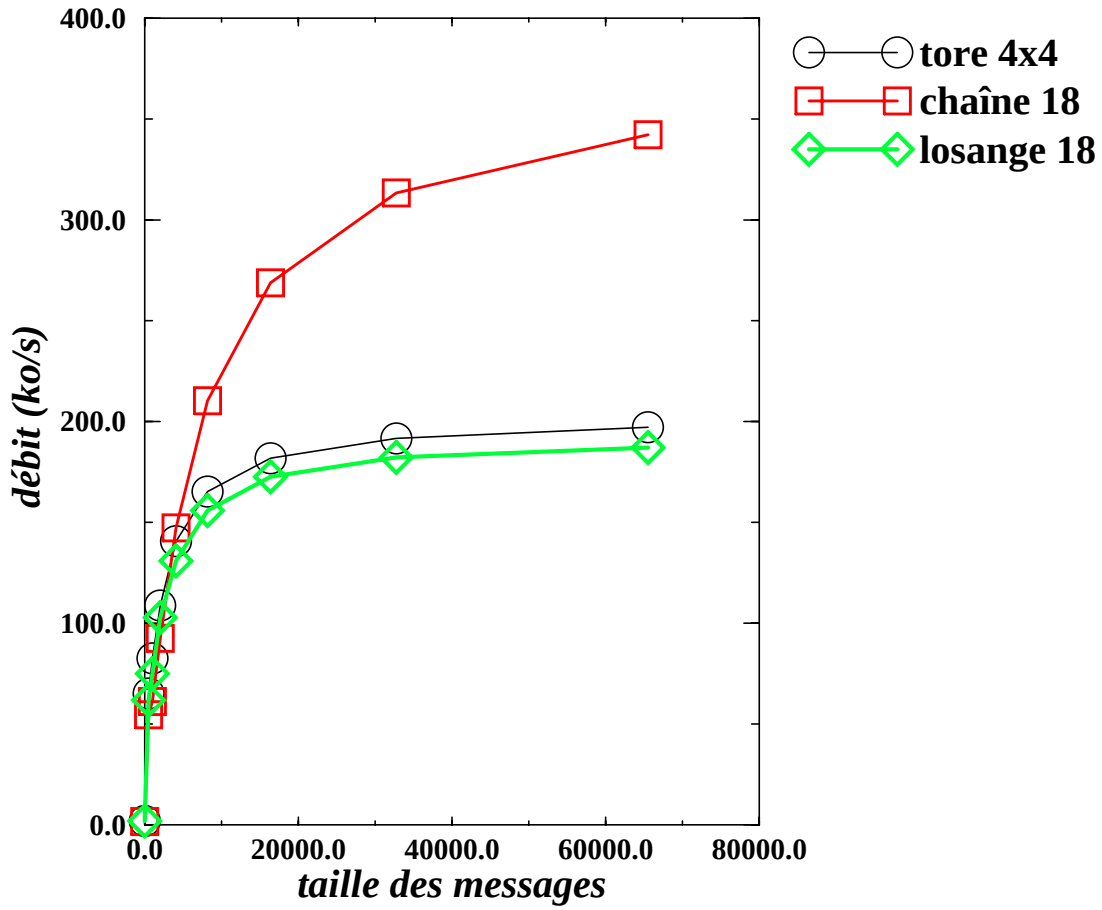


Figure 7.9 : impact de la topologie d'interconnexion.

7.6 Importance de la fonction de routage point-à-point

Tel que nous avons conçu notre machine virtuelle, le routeur à diffusion tient une place primordiale dans les performances du protocole de diffusion synchrone. Nous ne dégagerons pas ici la part prise par le routeur, mais nous allons nous intéresser à l'influence de la fonction de routage point-à-point, qui est la base de construction de l'algorithme de diffusion du routeur.

Nous reprenons ici les conditions générales de nos mesures, et ne faisons évoluer que la taille des messages avec les trois fonctions de routage disponibles dans ParX: l'arbre recouvrant et deux cycles eulériens. Les courbes de débits obtenues (cf. figure 7.10) nous montrent très clairement que les différences sont très faibles. Celles des deux cycles eulériens sont quasi-égales, et celle de l'arbre recouvrant n'est que très légèrement inférieure aux précédentes.

Il semble donc que pour une diffusion l'une ou l'autre des fonctions de routage point-à-point de ParX n'a pas une importance notable sur les performances de notre machine virtuelle. Mais comme nous l'avons remarqué au chapitre 5 §5.2.3, la différence se fait dans le comportement du routage des messages. Cette différence ne se manifeste pas dans les performances d'une même communication, mais devrait apparaître lorsque le système de communication est fortement chargé. Ceci s'explique, en corrélation avec la dernière remarque de la section 7.5, par l'utilisation des liens dans la fonction de routage, le degré des noeuds exploité ou l'existence de plusieurs chemins d'une source à une destinations.

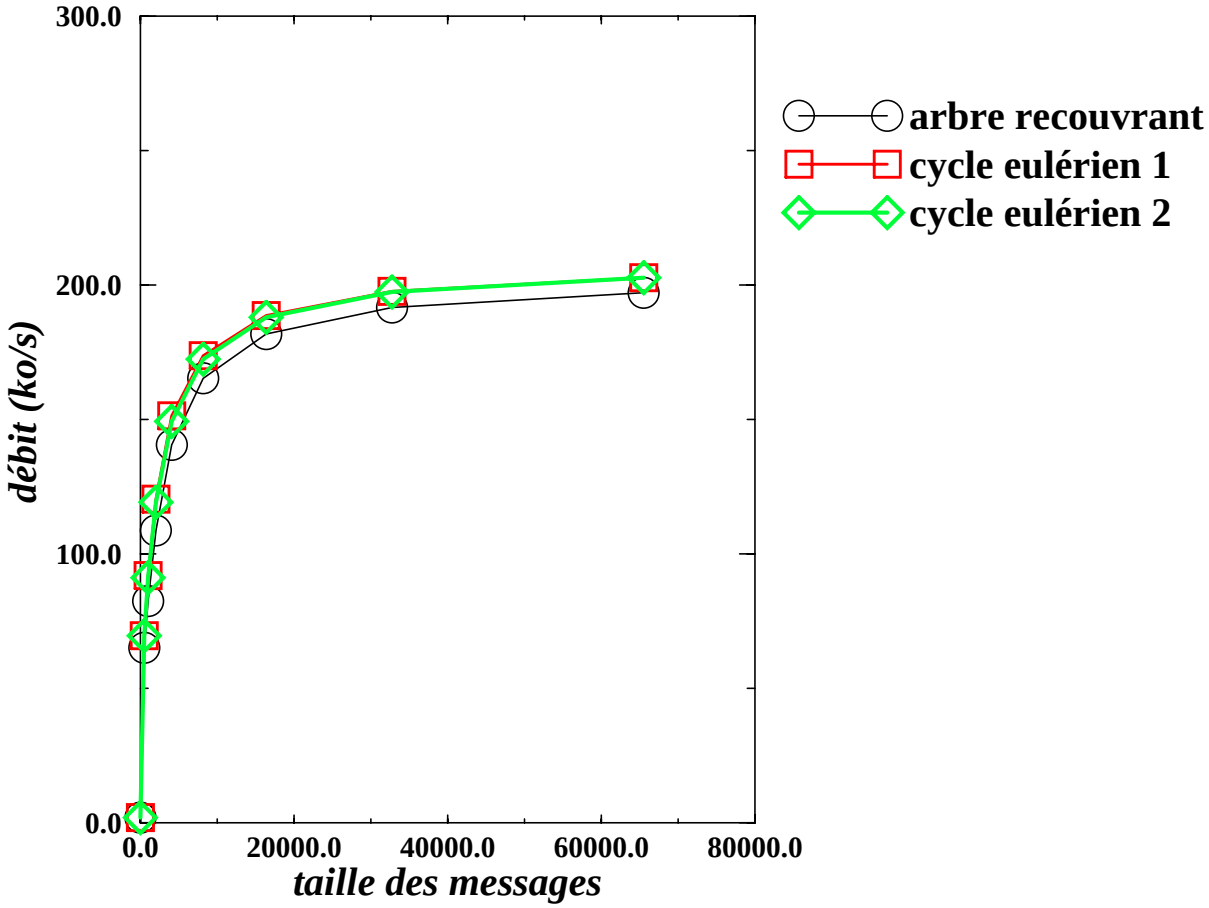


Figure 7.10 : importance de la fonction de routage point-à-point.

7.7 Comportement lors de diffusions simultanées

Cette fois le programme de mesure que nous avons employé ne comprend plus un seul émetteur, mais plusieurs dans des groupes différents. Nous avons fait évoluer à la fois le nombre d'expéditeurs (de 2 à 5) et leurs placements. Les figures 7.11 et 7.12 sont deux exemples caractéristiques des résultats obtenus pour deux et trois diffusions en parallèle.

Ce que nous pouvons noter de ces courbes c'est d'une part la perte de débit pour une diffusion et d'autre part la bonne répartition de la bande passante entre les diffusions concurrentes. D'une manière générale la perte de performances est proportionnelle au nombre de sites qui diffusent, et le débit pour une seule diffusion est divisé équitablement entre ces communications.

Toutefois, nous avons pu noter lors de nos tests que cette équité n'est pas toujours vérifiée et que certains noeuds sont plus privilégiés que d'autres. Mais nous pensons que ces exceptions sont plus dues à la fonction de routage point-à-point utilisée, l'arbre recouvrant, qui produit des arbres de diffusion plus optimaux pour ces noeuds, notamment pour la racine de l'arbre recouvrant.

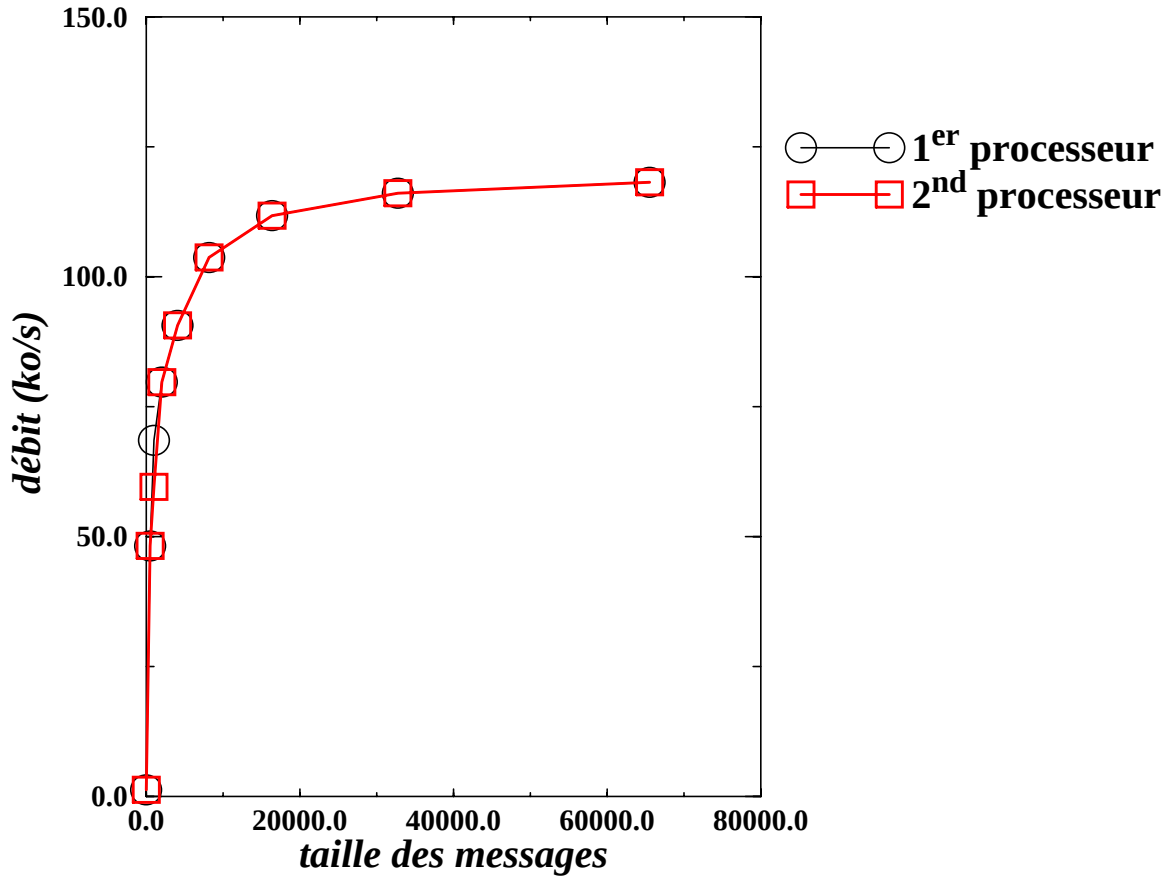


Figure 7.11 : débits pour deux diffusions simultanées.

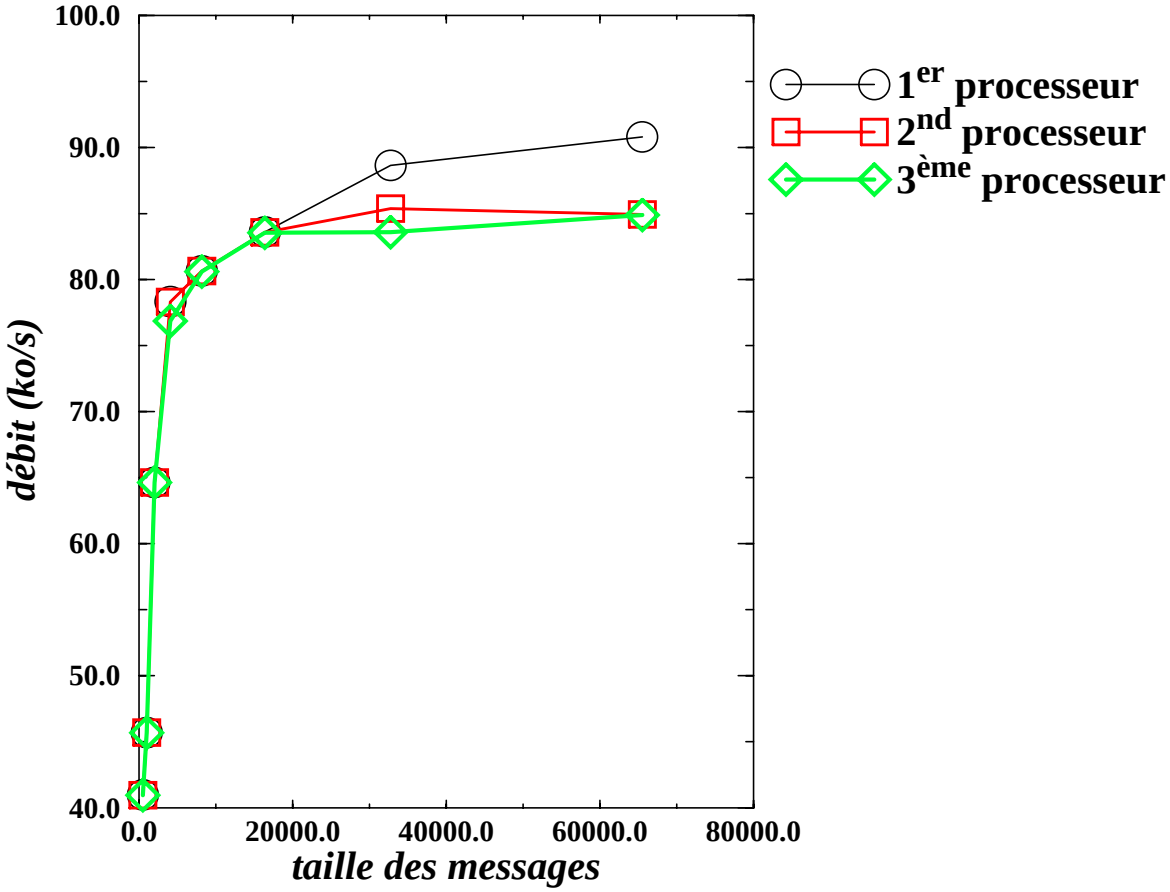


Figure 7.12 : débits pour trois diffusions simultanées.

7.8 Comparaison de la diffusion avec un échange point-à-point

Notre dernière analyse des performances du protocole de diffusion synchrone va nous permettre de le situer par rapport aux communications point-à-point. Pour cela nous avons comparé les résultats de la section 7.2, sur le tore 4x4 modifié, avec les mêmes mesures pour un protocole d'échange de message point-à-point. Afin que les résultats soient comparables nous avons considéré un protocole de sémantique équivalente, c'est-à-dire synchrone: en somme la réalisation distribuée du rendez-vous Occam offerte par ParX.

L'algorithme de mesure employé pour ce protocole est détaillé au paragraphe 7.1 et consiste à calculer, pour un processeur i, un temps total t_i de communication pour X messages de taille T pour chacun des N processeurs, y compris lui-même. A partir des mesures ainsi obtenues, nous avons déterminé de la manière suivante le débit moyen D:

$$D = \frac{X \times T \times N^2}{\sum_{i=0}^{N} t_i}$$

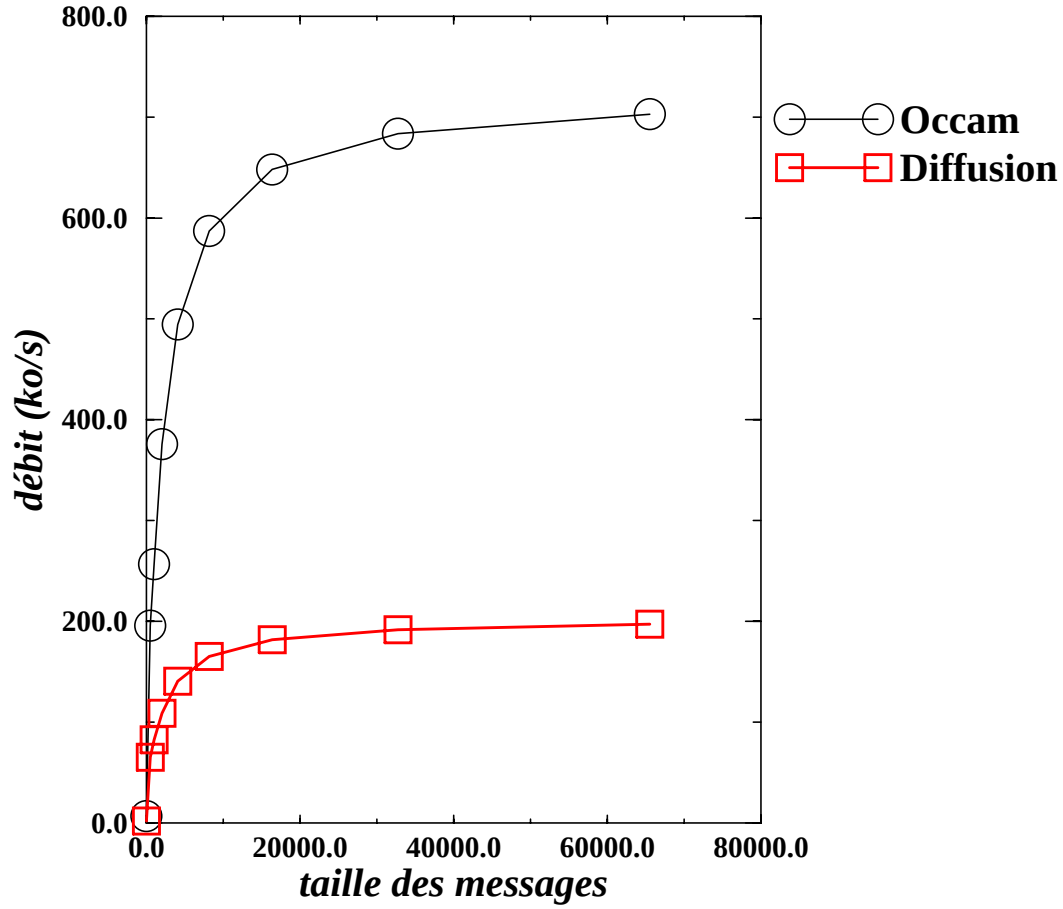


Figure 7.13 : comparaison de la diffusion avec un échange point-à-point.

Comme nous pouvons le constater sur la figure 7.13, le débit d'une diffusion est nettement plus faible que celui du protocole Occam, d'un facteur légèrement supérieur à 3. Ce qui est intéressant ce n'est pas d'observer cette différence évidente, mais de remarquer que le débit moyen de la diffusion (environ 200 ko/s) se situe entre le tiers et le quart de celui du protocole point-à-point (environ 700 ko/s); et donc qu'à partir de quatre destinations seulement le protocole de diffusion se montre nettement favorable. Pour atteindre l'ensemble des 18 processeurs à l'aide

du protocole Occam nous obtenons un débit d'un peu moins de 40 ko/s, au lieu des 200 ko/s du protocole de diffusion synchrone.

7.9 Conclusion

En résumé et de manière générale, nous pouvons dire que, pour les évaluations effectuées sur une machine reconfigurable de transputers T800, le débit de notre protocole de diffusion synchrone se situe aux alentours du cinquième de la bande passante des liens de communication, et ce quelle que soit la fonction de routage point-à-point sous-jacente, parmi celles disponibles avec ParX, et indépendamment de la topologie d'interconnexion, du nombre de processeurs ou du site émetteur. Nous noterons également que ce débit est équitablement divisé lors de diffusions simultanées, même s'il peut apparaître des noeuds plus privilégiés.

Par ailleurs, puisque le transputer intègre un dispositif d'accès direct à la mémoire (DMA) et d'une instruction spéciale de copie de zones mémoires, nous pouvons négliger le coût des copies mémoires effectuées par le protocole. Le coût du protocole est donc essentiellement celui de la synchronisation, soit celui de la diffusion d'un message de très petite taille (8 octets par exemple): 2 ko/s. Dans ces conditions, le débit de l'algorithme de routage pour la diffusion est approximativement celui du protocole synchrone de PDVM pour des messages de taille importante, soit environ 200 ko/s où le cinquième de la bande passante maximale du réseau.

Chapitre 8 : Conclusion et Perspectives

Au début de ce manuscrit nous avons présenté les divers outils du parallélisme. Des différents types de machines abordés nous avons retenu, comme cadre de nos travaux, les architectures massivement parallèles qui, parce qu'elles sont extensibles, permettent de dimensionner une machine suivant la taille du problème à traiter. Le problème majeur de ces architectures est qu'elles n'offrent pas physiquement une mémoire partagée; c'est-à-dire que l'échange de messages est le seul véritable paradigme de communication, de synchronisation et de coopération qu'elles exploitent. C'est là le centre de nombreuses recherches, tant autour des communications point-à-point qu'autour de schémas plus complexes comme les communications globales. C'est à ce dernier domaine que notre thèse propose une solution nouvelle.

Une étude générale des outils de développement et d'exécution pour le parallélisme, langages et environnements, outre le fait de situer nos travaux, nous a tout d'abord démontré l'intérêt des communications globales et de la diffusion. Cette étude conjointe avec celle des systèmes d'exploitations adaptés aux ordinateurs parallèles nous montrent de plus la nécessité de se placer dans le cadre d'un système d'exploitation, et en particulier PAROS/ParX conçu et réalisé par notre équipe.

Ainsi orientés, nos travaux ont été décomposés en deux parties. Il a s'agit dans un premier temps d'apporter une solution au problème de la diffusion dans un réseau quelconque, c'est-à-dire au niveau des processeurs. Puis nous nous sommes investis dans la conception et le développement d'un support d'implantation de tout système à diffusion, afin d'offrir aux processus des protocoles généraux de communication par diffusion.

8.1 Contribution au routage

Bien que la finalité de notre travail dans ce domaine n'était pas d'aboutir à la fabrication d'un véritable circuit, notre objectif était malgré tout de proposer des solutions à la diffusion capables d'être intégrées à un routeur matériel et indépendantes du réseau, du point de vue de sa taille et de sa topologie d'interconnexion.

Pour cela nous nous sommes tout d'abord inspiré des recherches menées sur les communications point-à-point et les solutions qui se montrent les plus intéressantes. Puis, de l'étude des différentes techniques de diffusion proposées, sous l'angle d'une classification originale, nous avons pu constater qu'aucune d'entre elles ne satisfaisait l'ensemble des contraintes liées à notre objectif.

Notre choix s'est tout de même porté sur l'une de ces méthodes: l'inondation, pour son indépendance vis-à-vis du réseau d'interconnexion, ainsi que pour sa flexibilité qui la rend évolutive. Deux obstacles restaient à surmonter: avant tout la possibilité de génération d'interblocages, puis les forts trafics et encombrements générés par le grand nombre de duplications inutiles d'un même message.

La représentation du problème des interblocages nous a conduit à une nouvelle technique d'inondation qui consiste à conditionner l'utilisation des liens du réseau par les interdictions imposées par la fonction de routage point-à-point. Bien plus que de rendre la technique d'inondation correcte, nous avons montré que de surcroît notre algorithme devenait également paramétrable et que, selon la fonction de routage point-à-point employée, le comportement de la diffusion variait pour produire des temps et des encombrements différents. Ainsi, avec une fonction de routage déterministe, l'élimination des duplications inutiles devient superflue.

Afin de rendre notre algorithme plus proche des contraintes imposées par une intégration dans un circuit routeur nous l'avons adapté au routage par intervalles. Enfin nous avons proposé une architecture d'un tel routeur, dont les algorithmes des interfaces des liens d'entrée et des arbitres des liens de sortie ont été détaillés.

8.2 Contribution aux systèmes à diffusion

Forts des résultats obtenus dans le routage pour la diffusion nous nous sommes ensuite intéressés à la conception et à la réalisation d'une machine virtuelle à diffusion pour le système PAROS/ParX. Notre objectif principal n'était pas de définir un nouveau système à diffusion mais de construire un support d'implémentation pour la plupart des interfaces de communication de groupe.

Si l'étude des principaux environnements de programmation fondés sur l'échange de message nous a montré l'importance croissante des communications collectives, celle des systèmes plus spécifiquement centrés autour de la diffusion nous a permis de dégager les divers protocoles nécessaires et fonctions essentielles. Ainsi, pour qu'elle soit compatible avec le plus grand nombre de ces systèmes, nous avons inclu à notre machine virtuelle, PDVM, deux protocoles de diffusions: l'un synchrone et le second asynchrone; ce dernier dispose en outre de deux sémantiques selon la présence ou l'absence de tampons.

Pour cette raison de compatibilité, PDVM n'intègre pas de véritable gestion des groupes de processus, gestion qui s'est avérée nécessaire et qui est le point le plus divergent entre toutes les interfaces. Il s'agit là que d'un support minimal pour la manipulation de groupes, mais qui garantit toutefois la cohérence des communications vis-à-vis d'une telle structure: un message diffusé à un groupe n'est reçu que par les membres qui appartiennent au groupe au moment ou se produit la communication, et seulement par eux. De même l'interface d'accès aux protocoles est réduite à un petit ensemble de primitives, à partir desquelles des fonctions plus évoluées peuvent être implémentées.

Le tout est intégré dans une structure particulière qui bénéficie de l'architecture du micro-noyau ParX : le protocole synchrone est implanté génériquement dans le micro-noyau, au-dessus du routage à diffusion offert par le HEM; et le protocole asynchrone au-dessus du protocole synchrone, toujours dans ParX mais en dehors du micro-noyau. Nous avons d'ailleurs pu remarquer l'efficacité, à la fois de ce choix de réalisation et de l'architecture générale de ParX.

8.3 Continuation de nos travaux

Comme suite naturelle à notre thèse dans ce domaine, nous pensons avant tout à l'intégration dans un circuit de l'algorithme qui se montrera le mieux approprié, parmi ceux proposés. Ceci passe tout d'abord par l'évaluation des diverses solutions de routeurs exposées dans ce manuscrit, afin de retenir la mieux appropriée à cette implémentation matérielle tant du point de vue de l'efficacité qu'en ce qui concerne la complexité du routeur.

Nous pensons également que les travaux de recherches que nous avons effectués sur le routage pour la diffusion devrait être poursuivies pour d'autres schémas de communications. Ce peut être des échanges de messages plus généraux de type *n* vers *p*. Mais ce peut être aussi des échanges de type *n* vers 1, dont nous avons pu constater l'impact sur le coût de notre protocole de diffusion synchrone.

Finalement, en conjonction avec les résultats obtenus dans le routage, nous noterons qu'un groupe devrait bénéficier du routage par intervalles pour représenter la distribution de ses membres sur le réseau. De plus cela ouvre des perspectives nouvelles au problème du placement des processus et répartition de charge. En effet, puisque les communications de groupe induisent des trafics élevés de messages, la réduction du champ de ces échanges dans le réseau apporterait un gain notable de performances.

Annexes

Annexe A : Langages Parallèles

Notre intention ici n'est pas de couvrir exhaustivement l'ensemble des langages qui sont disponibles, mais de présenter ceux qui nous paraissent les plus représentatifs. En plus des différences traditionnelles dans l'expression du problème à traiter (de manière impérative ou déclarative), les langages parallèles se distinguent selon qu'ils ont été conçus comme des extensions de langages (C, FORTRAN, etc.), comme des langages concurrents (**Ada**), ou directement comme des langages parallèles (**Occam**). Les trois cas seront abordés ici. Les langages déclaratifs et fonctionnels tels que LISP ou PROLOG étant intrinsèquement parallèles nous ne les aborderons pas dans cette annexe. Nous renvoyons le lecteur à [Hast85] pour **Multilisp** ou [Shap83] pour **Concurrent Prolog**.

A.1 Extensions de langages

Partir sur les bases d'un langage déjà connu pour aborder la programmation parallèle est une démarche qui séduit bon nombre d'utilisateurs; ainsi ils peuvent concentrer leur attention à l'apprentissage de l'algorithmique parallèle et des quelques primitives et constructions syntaxiques supplémentaires. D'autres raisons peuvent justifier d'adapter un langage séquentiel au parallélisme, par exemple disposer du langage C peut s'avérer plus commode pour programmer des services systèmes. Bref, argument promotionnel ou volonté d'offrir une panoplie complète d'outils, les distributeurs de langages proposent des versions de langages adaptées à un grand nombre de machines parallèles.

Pour être représentatif nous avons choisi de faire une synthèse de quelques-unes des versions parallèles de quatre langages célèbres: FORTRAN, MODULA-3, C++ et C.

A.1.1 FORTRAN parallèle

Historiquement c'est le calcul scientifique qui fournit le plus vaste ensemble d'applications pour le parallélisme. Il est donc compréhensible que FORTRAN, langage quasi-unique de la communauté du calcul scientifique, ait évolué pour s'adapter aux super-calculateurs du marché.

Pour cette évolution, et parce que les architectures SIMD ont connu un plus net succès que les machines MIMD auprès des scientifiques, le FORTRAN parallèle est plus particulièrement destiné au parallélisme de données. Aujourd'hui cependant, **High Performance FORTRAN** (**HPF**) [HPF93] est disponible non seulement pour les calculateurs SIMD, mais aussi pour les machines MIMD. HPF, que nous avons choisi de présenter ici, est issu des efforts conjoints de nombreux partenaires, développeurs et utilisateurs de part le monde, et peut être considéré comme une norme.

Les extensions essentielles apportées au langage sont en fait peu nombreuses, et assez simples à comprendre. Héritier du FORTRAN 90 [Merl92] et du FORTRAN D [FHK91], HPF reprend les modifications qu'ils ont respectivement effectués pour le traitement des tableaux, et pour leur distribution sur la machine.

Jusqu'alors, dans les versions antérieures au FORTRAN 90, les éléments d'un tableau ne pouvaient être accédés que les uns après les autres, désormais il est possible de manipuler directement tous les éléments d'un tableau, ou une portion d'un tableau. Appliquer une même modification à tout un ensemble d'éléments est de cette façon possible et ne nécessite plus de structure répétitive. Ainsi la boucle suivante:

```
REAL V(100)
...
DO I = 1,100
    V(I) = 10 * V(I)
END DO
```

peut s'écrire en une seule expression:

```
V = 10 * V
```

De même il est possible d'effectuer certaines opérations complexes sur un tableau, par exemple:

```
V = SIN(V)
```

Les portions de tableaux sont accessibles avec la même facilité, par exemple l'affectation du vecteur `V` à la troisième colonne d'une matrice `M` (`REAL M(100,100)`) se fait très simplement:

```
M(3,1:100) = V
```

ce qui est équivalent à:

```
M(3,:) = V
```

où «`:`» est l'abréviation de la dimension entière. De plus ces sous-ensembles peuvent définir des éléments non consécutifs, comme le montre l'instruction suivante qui affecte les éléments d'indices pairs de `V` aux 50 derniers éléments de la première colonne de `M`:

```
M(1,50:100) = V(2:100:2)
```

Les décalages sont tout aussi aisément exprimés, tel que nous le montre l'opération suivante:

```
V(1:98) = V(2:100)
```

L'apport de FORTRAN D est d'offrir une expression des alignements de tableaux, les uns par rapport aux autres, adaptés au problème à traiter, et, un moyen d'effectuer une distribution ad hoc. Trois types de distributions sont possibles: **BLOCK**, **CYCLIC**, et **BLOCK_CYCLIC**. Si l'on considère K processeurs et un vecteur V de taille N, la distribution BLOCK décompose V en K blocs de N/K éléments consécutifs; puis affecte au processeur P_i le bloc V((i-1)N/K+1:iN/K). Par contre en opérant une distribution CYCLIC les éléments de V sont affectés tour à tour aux processeurs de telle sorte que le processeur P_i reçoive le bloc V(i:N:K). Combinaison des deux, la variante BLOCK_CYCLIC applique la distribution CYCLIC, non plus sur des éléments, mais sur des blocs d'éléments consécutifs. La figure A.1 illustre ces trois méthodes de distribution, avec N égal à 10, P égal à 3, et une taille de bloc de 2 pour BLOCK_CYCLIC.

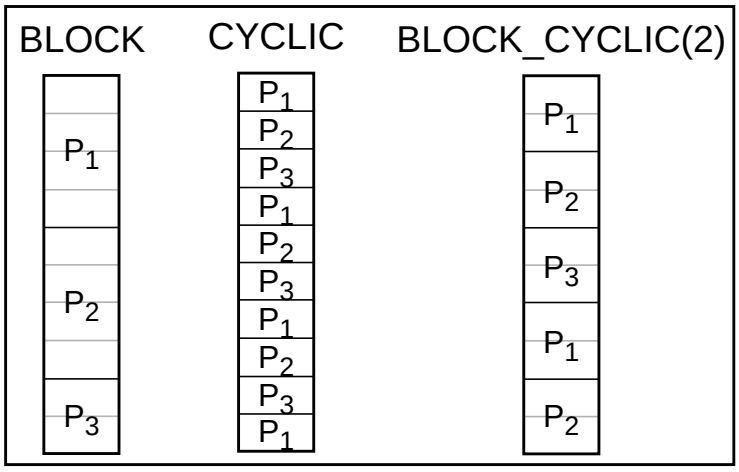


Figure A.1 : distributions d'un vecteur en FORTRAN D.

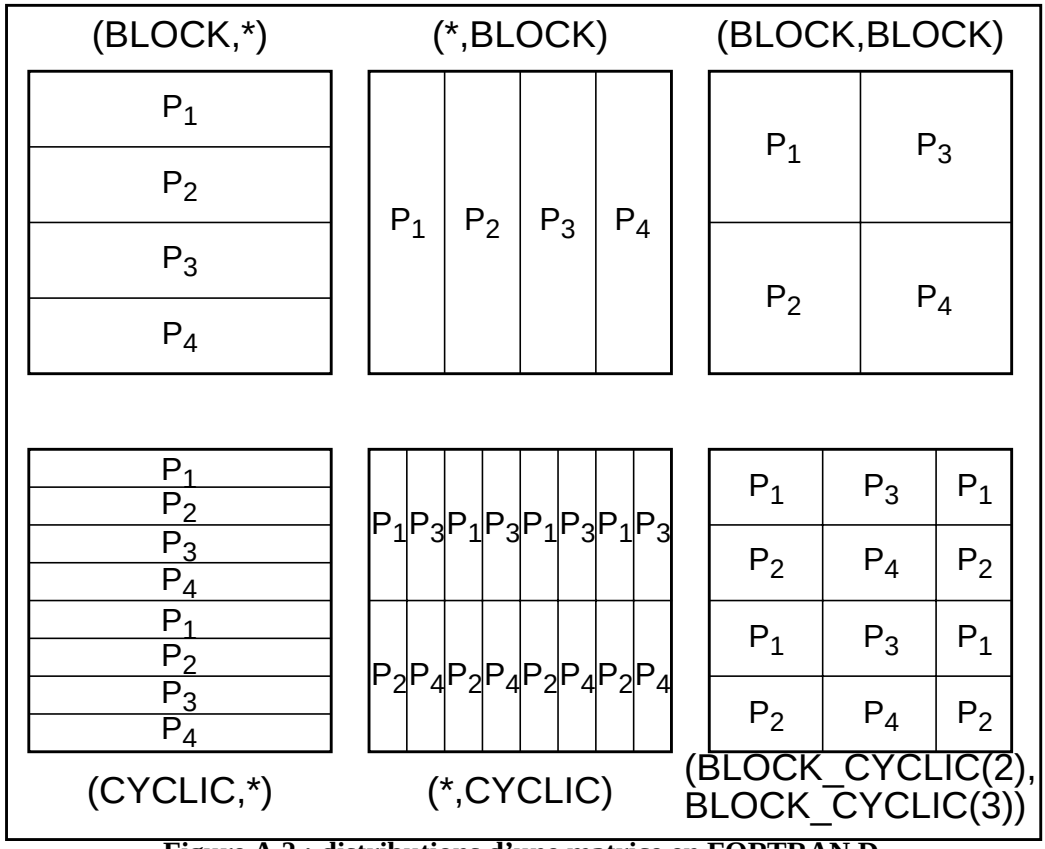


Figure A.2 : distributions d'une matrice en FORTRAN D.

Le schéma de distribution des données d'un vecteur se généralise, dimension par dimension, aux tableaux multi-dimensionnels. La figure A.2 donne un aperçu des possibilités de distribution d'une matrice sur quatre processeurs.

Une fois distribués, pour aligner les éléments de divers tableaux sur un même noeud (processeur et/ou mémoire), il faut définir un objet de décomposition des tableaux. Cette décomposition se présente sous la forme d'un tableau qui n'occupe pas d'espace de stockage et permet, en quelque sorte, de désigner un placement identique des données. L'alignement est en fait effectué élément par élément, de manière telle qu'aucune communication ne soit requise pour effectuer des calculs sur des éléments alignés. Par exemple dans le programme suivant, et quelle que soit la distribution des données, l'opération s'effectuera sans communication:

```
REAL V1(N), V2(N)
DECOMPOSITION D(N)
ALIGN V1, V2 with D -- aligne V1(I) et V2(I) avec D(I)
...
V1 = 10 * V2
```

Calculer la transposée d'une matrice peut se faire également sans avoir recours à des communications, à partir de l'alignement suivant:

```
REAL M(N,N), T(N,N)
DECOMPOSITION D(N,N)
ALIGN M(I,J), T(J,I) with D(I,J)
```

Il en va de même pour le calcul de la diagonale d'un produit de matrices, lorsque les lignes de la première matrice sont alignées avec les colonnes de la seconde:

```
REAL M1(N,N), M2(N,N), P(N,N)
DECOMPOSITION D(N)
ALIGN P(I,I), M1(I,J), M2 (J,I) with D(I)
```

Outre l'alignement et la distribution des données, FORTRAN D offre une nouvelle construction syntaxique qui permet de contrôler l'exécution d'itérations: **FORALL**. Le contrôle s'opère à la fois sur les effets de bord des itérations entre elles et sur le placement des calculs par rapport à celui des données. Avec FORALL aucune modification de variable n'est répercutée sur les autres itérations qui, par conséquent, n'accéderont qu'à l'ancienne valeur de la variable. Pour influer sur le placement d'un calcul l'utilisateur a la possibilité de spécifier le processeur par rapport à l'une ou l'autre des variables manipulées: mot clef **HOME**. Par exemple dans le programme ci-après, chaque itération s'effectue sur le processeur ou doit être enregistré le résultat, et les nouvelles valeurs des X(I+10) sont calculées par rapport aux anciennes valeurs des X(I).

```
REAL X(50), Y(40)
...
FORALL I = 1,40 on HOME(X(I+10))
    X(I+10) = F(X(I),Y(I))
ENDFOR
```

Nous remarquerons que ces évolutions de FORTRAN permettent d'éviter aux développeurs l'expression des communications. Celles-ci sont alors mises en oeuvre, de manière transparente, par le support d'exécution. A ce niveau, la diffusion peut s'avérer fort utile. Par exemple pour la multiplication par une constante d'un vecteur distribué, il est plus naturel de diffuser la constante à chacun des processeurs qui dispose d'une partie du vecteur.

A.1.2 MODULA-3* et MODULA-3π

L'approche adoptée dans MODULA-3* [Heinz93] est de rendre l'expression du parallélisme facile d'accès et centrée sur le problème. Pour cela seulement deux constructeurs syntaxiques nouveaux ont été ajoutés au langage de base: **FORALL** et **NOSYNC**. Le premier permet d'exprimer la composition parallèle de calculs dans une syntaxe similaire à celle d'un FOR ordinaire. Tous les processus créés par FORALL ont le même code, mais en général ne traitent pas les mêmes données. Chacun de ces processus est identifié par la valeur des constantes qui sont utilisées dans la construction. A titre d'exemple, le calcul de la transposée d'une matrice carrée s'écrit de la manière suivante:

```
FORALL i,j:[1..n] DO
    M[i,j] := M[j,i];
END
```

Ici, comme en FORTRAN D, le problème d'interférence entre processus ne se pose pas et il ne peut y avoir d'effet de bord: tous les processus accèdent aux anciennes valeurs contenues dans M, et non à celles nouvellement calculées. Ceci vient du fait du synchronisme imposé aux processus qui se rapproche du mode SIMD. Cependant l'appel de procédure provoque une rupture de synchronisation, et seul le passage de paramètres reste synchrone. Par contre, qu'il y ait ou non rupture, la terminaison des processus est toujours synchrone.

Le langage permet également d'exprimer un fonctionnement asynchrone pour une étape intermédiaire ou pour tout un calcul. Ceci se fait en englobant dans des blocs particuliers, identifiés par le mot clé NOSYNC, chaque partie asynchrone. La terminaison (END) de chacun de ces blocs d'instructions marque la reprise du mode synchrone. Dans l'exemple suivant, la première étape qui consiste à multiplier le vecteur V par 10 ne nécessite pas une exécution synchrone; alors que la seconde qui va effectuer la permutation p sur V en a besoin.

```
FORALL i:[1..n] DO
    NOSYNC
        V[i] = 10 * V[i];
    END;
    (* ici les processus sont resynchronisés *)
    V[i] := V[p(i)];
END
```

Il est possible de simplifier l'écriture lorsque tous les processus de la construction FORALL peuvent entièrement s'exécuter indépendamment les uns des autres, comme c'est le cas dans l'addition de vecteurs:

```
FORALL i:[1..n] NOSYNC DO
    R[i] = V1[i] + V2[i];
END
```

Par ailleurs, il n'y a aucune restriction dans l'appel de procédure et toutes les procédures visibles peuvent être invoquées; ce qui est possible car tous les processus créés partagent le même espace d'adressage.

Comme pour FORTRAN, si l'on se pose le problème de l'efficacité il faut prendre en considération l'alignement des données et des calculs. C'est ce qui est offert avec les extensions de MODULA-3π qui proposent un constructeur syntaxique d'alignement: **ALIGN**, et trois types de distribution de données: **LOCAL**, **SPREAD** et **CYCLIC**.

ALIGN peut être utilisé soit pour indiquer le placement d'un calcul par rapport à une donnée, soit pour aligner les données les unes par rapport aux autres. Le placement d'un processus se fait sur le processeur qui dispose de la donnée que l'utilisateur spécifie comme base de l'alignement des calculs. Ceci est valable autant pour une partie séquentielle que pour une composition parallèle. Sur l'exemple de l'addition de vecteurs chaque somme d'éléments pourrait être effectuée par le processeur où le résultat sera récupéré:

```
FORALL i:[1..n] NOSYNC DO
    ALIGNED WITH R[i] DO
        R[i] = V1[i] + V2[i];
    END;
END
```

Pour augmenter encore l'efficacité lors de l'exécution d'un tel code, il faut ajuster l'alignement des données de façon à ce que, pour chaque processus, les données qui interviennent dans les différents calculs soient directement disponibles, sans communication, pour le processeur auquel est affecté l'exécution du processus. Dans notre précédent exemple cela se fait très simplement par alignement des éléments de même indice:

```
ALIGN V1, V2 WITH R;
```

De manière générale l'alignement des données est plus sophistiqué et correspond aux accès qui sont exprimés dans les calculs. L'alignement suivant illustre bien les possibilités offertes par MODULA-3π:

```
FORALL i:[1..n]
    ALIGN X[i+1], Y[2*i] WITH Z[i,i];
```

De plus, un peu comme les décompositions de FORTRAN D, les expressions employées peuvent se référer à des types de tableaux préalablement définis.

La distribution quant à elle s'exprime lors de la déclaration des tableaux et pour chacune des dimensions; par défaut c'est la méthode `LOCAL` qui est utilisée. Celle-ci affecte l'ensemble de la dimension à un même processeur. Une allocation identique à celle du type `BLOCK` de FORTRAN D est effectuée lorsque `SPREAD` est spécifié. Enfin la distribution `CYCLIC` est équivalente à la combinaison `BLOCK_CYCLIC` de FORTRAN D.

La proximité des concepts d'expression du parallélisme de MODULA-3* et MODULA-3π avec ceux de HPF, rend les communications également transparentes. Ainsi comme précédemment, la diffusion pourrait être une opération bénéfique pour le support d'exécution de tels programmes.

A.1.3 CC++

Un programme CC++[1] [CCK93] est un programme C++ qui se compose d'un ou plusieurs objets *processeur*. La déclaration de tels objets est similaire à celle d'une classe ordinaire; le mot clé **`global`** permet au compilateur de faire la distinction:

```
global Proc {
    // déclaration des structures de données et méthodes
}
```

Comme pour toute autre classe, les données et méthodes publiques sont accessibles à l'extérieur de l'objet. Cependant les restrictions imposées par CC++, interdisent l'héritage à partir d'un objet processeur, la déclaration de membres protégés, de fonctions virtuelles ou de données statiques. Ils peuvent par contre être créés dynamiquement en utilisant l'opérateur **`new`**.

L'exécution d'un programme est contrôlée par un objet processeur initial: le programme démarre lors de la création de cet objet et se termine avec la destruction de celui-ci. Cet objet indispensable doit contenir une fonction **`main`** publique, et il doit être le seul. Comme pour un programme C, arriver au terme de cette fonction se traduit par la terminaison du programme; ici cela se fait par la destruction de l'objet initial. La primitive **`exit`** dont le but est de mettre fin à l'exécution de l'application procède de la même façon, c'est-à-dire qu'après avoir mis fin à l'exécution de tout objet processeur, elle appelle le destructeur de l'objet initial.

Trois opérateurs de composition parallèle sont définis: **`par`**, **`parfor`**, et **`spawn`**. Le premier permet d'exécuter en parallèle une liste d'instructions:

```
par {
    // instructions à exécuter en parallèle
}
```

1. cCompositional C++

L'exécution d'un `par` prend fin lorsque tous ses composants ont terminé la leur; il s'agit donc, en quelque sorte du pendant de la composition séquentielle. Toutefois il est impossible d'utiliser les flots de contrôle créés pour effectuer des calculs en dehors du bloc même. De ce fait les instructions **`break`**, **`continue`** ou **`return`**, qui provoqueraient la sortie inopinée du bloc, sont interdites.

Tout comme `par`, `parfor` est le pendant des itérations séquentielles exprimées avec `for`, et pour les mêmes raisons, ce qui n'est pas admis avec `par` ne l'est pas non plus avec `parfor`. Si ces deux compositions ont la même syntaxe, les itérations ne sont plus ici évaluées séquentiellement, mais de manière concurrente.

Contrairement aux deux constructeurs précédents qui suspendent l'évaluation de la suite d'instructions dans laquelle ils apparaissent, `spawn` permet d'exécuter des processus en parallèle avec le processus qui les crée. L'utilisation de `spawn` s'apparente à l'appel de fonction, mais ici la valeur retournée est toujours éliminée.

Enfin, pour contrôler la concurrence entre les divers processus et effectuer des synchronisations, l'utilisateur dispose d'un nouveau type de fonctions: **`atomic`**, et d'un nouveau type d'objets: **`sync`**. L'exécution d'une fonction de type `atomic` ne peut être interrompue et permet donc d'effectuer des opérations en exclusion mutuelle. Par ailleurs un objet de type `sync` est créé dans un état non initialisé, et ne peut être initialisé qu'une et une seule fois. Dans un état non initialisé toute tentative de lecture de cet objet est suspendue jusqu'à ce qu'une valeur lui soit affectée; il est ainsi possible de synchroniser des processus.

Cette évolution de C++ qui permet le placement des objets et leur parallélisation, suit le mode de communication propre au modèle de programmation par objets. Ainsi seules les communications point-à-point sont requises pour réaliser l'invocation de méthodes, et la diffusion n'est donc pas directement nécessaire.

A.1.4 C Inmos

Contrairement aux cas précédents rien n'a été ajouté au langage de base; ce sont la chaîne de compilation et une bibliothèque de fonctions et structures de données qui permettent d'exploiter le parallélisme. Avec ce compilateur [Inmos90a, Inmos90b], conçu pour la famille des transputers, l'utilisateur peut garder un contrôle de bas niveau, possibilité qui est à l'origine du langage C et qui en fait un langage privilégié pour le développement des systèmes d'exploitation.

Une application parallèle est ici un simple ensemble de programmes C auquel est adjoint une description des processeurs, du réseau d'interconnexion et du placement des programmes. L'obtention d'un fichier exécutable, en plus de la compilation et de l'édition de liens, passe par deux étapes supplémentaires qui insèrent aux codes des processeurs les informations relatives à la configuration et nécessaires au chargement de l'application sur le réseau. Nous avons alors une chaîne de compilation qui comprend un compilateur: **icc**, un éditeur de lien: **ilink**, un «configureur»: **icconf**, et un «collecteur» de code: **icollect** (voir figure A.3). A cela viennent s'ajouter d'autres outils pour gérer des bibliothèques, pour la mise au point et la simulation, ces deux derniers étant assez rudimentaires par rapport à la complexité de la programmation parallèle.

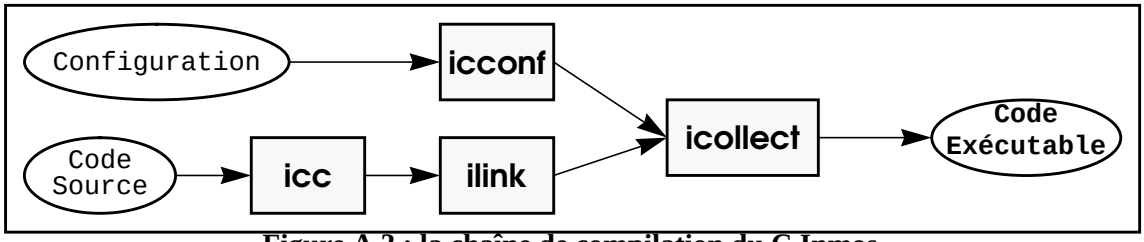


Figure A.3 : la chaîne de compilation du C Inmos.

La description des processeurs, du placement des programmes et de la connectique se fait dans un langage spécialisé et simple. La déclaration d'un ou plusieurs processeurs se fait en indiquant leur `type` et la quantité de mémoire dont ils disposent:

```
processor (type = ”T414”, memory = 512k) M;
processor (type = ”T800”, memory = 1M) P[4];
```

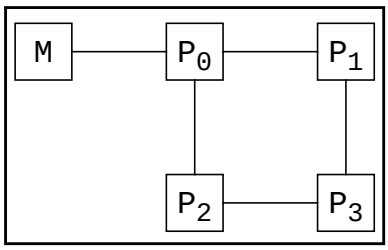


Figure A.4 : grille 2x2 avec processeur hôte.

A partir des processeurs définis l'utilisateur décrit son réseau d'interconnexion; par exemple pour reproduire la «grille» de la figure A.4, il suffit de décrire les connexions physiques employées:

```
connect M.link[0] to P[0].link[0];

connect P[0].link[1] to P[1].link[0];
connect P[0].link[2] to P[2].link[0];
connect P[1].link[1] to P[3].link[0];
connect P[2].link[1] to P[3].link[1];
```

Le placement du code sur les processeurs se fait en trois étapes, tout d'abord il faut déclarer les processus, c'est-à-dire les environnements d'exécution des différents programmes, avec pour chacun d'eux une pile, un tas et une interface paramétrable. Dans l'exemple suivant sont déclarés un processus maître et un tableau de quatre processus esclaves tous identiques:

```
process (stacksize = 10k, heapsize = 250k,
         interface(int master = 1)) MasterProc;

process (stacksize = 5k, heapsize = 500k,
         interface(int master = 0)) SlaveProc[4];
```

Ensuite les programmes de l'application, obtenus après l'édition de lien (format `.lku`), sont associés à ces processus:

```
use ”master.lku” for MasterProc;
rep i = 0 to 3
    use ”slave.lku” for SlaveProc[i];
```

La dernière étape consiste à placer les processus sur les processeurs:

```
place MasterProc on M;
rep i = 0 for 4
    place SlaveProc[i] on P[i];
```

Il s'agit donc là d'un outil minimal d'allocation statique de processus. Comme c'est à l'utilisateur que revient la décision de l'allocation cela en fait un outil mal adapté au parallélisme massif, où la quantité de processeurs et de processus est humainement difficile à maîtriser.

Dans la programmation des applications la concurrence est exprimée par un ensemble de primitives et structures de données disponibles à partir de la bibliothèque standard du C Inmos. Ainsi la création ou la destruction de processus, la synchronisation et la communication se font par appels de fonctions. En revanche l'utilisateur doit gérer lui-même les structures de données nécessaires à ces fonctions; il a ainsi un contrôle de bas niveau de l'expression du parallélisme.

Hormis les processus définis lors de la configuration de l'application (icconf), toute autre exécution de processus se fait en deux étapes. En premier lieu il faut allouer et initialiser la pile et le contexte de travail puis associer au processus sa procédure principale; ceci se fait à partir des deux fonctions **`ProcAlloc`** et **`ProcInit`**:

```
Process *ProcAlloc(
    void (*Fonction)(), /* adresse fonction à exécuter */
    int TaillePile,
    int NombreParametres,
    ... /* paramètres du processus, i. e. de la fonction */)

int ProcInit(
    Process *Processus,
    void (*Fonction)(),
    int *Pile, /* zone mémoire déjà allouée pour la pile */
    int TaillePile,
    int NombreParametres,
    ... /* paramètres du processus, i. e. de la fonction */)
```

Ce n'est qu'une fois obtenues les structures données associées aux processus que son exécution est possible. Le démarrage d'un nouveau processus peut se faire de manière concurrente et indépendamment des autres processus grâce à la primitive **`ProcRun`**. De plus l'exécution coordonnée d'un ensemble de processus, à l'image de la sémantique du constructeur parallèle de CSP, se fait soit à l'aide de la fonction **`ProcPar`**, soit à l'aide de **`ProcParList`**. Dans ce dernier cas le processus appelant reste bloqué jusqu'à la terminaison de tous les processus spécifiés dans l'appel, et ces processus démarrent et prennent fin simultanément.

Suivant le même modèle, l'implémentation des sémaphores impose une allocation ou initialisation: **`SemAlloc`** et **`SemInit`**. Les opérations usuelles, **`SemWait`** et **`SemSignal`**, sont ensuite utilisables.

En plus des sémaphores les processus peuvent communiquer par échanges de messages au travers de canaux unidirectionnels. L'implémentation de ces canaux respecte le modèle des processus communicants de CSP, c'est-à-dire qu'il y a *rendez-vous* entre les processus lors des communications. Les fonctions de manipulation des canaux sont: **`ChanAlloc`** et **`ChanInit`** pour l'initialisation, **`ChanIn`** pour l'émission, et **`ChanOut`** pour la réception.

Par ailleurs, en réception, un processus peut se mettre en attente sur plusieurs canaux. L'appel à **`ProcAlt`** ou **`ProcAltList`** bloque le processus jusqu'à ce que l'un au moins des canaux spécifiés soit prêt, c'est-à-dire lorsque le rendez-vous peut s'établir. Ces fonctions retournent le numéro de l'un des canaux prêts, mais en aucun cas elles n'effectuent la communication, que ce soit entièrement ou partiellement. Dans ce cas l'attente est non bornée et seule l'émission sur l'un des canaux peut réveiller le processus. Pour limiter cette attente la bibliothèque dispose des primitives **`ProcTimerAlt`** et **`ProcTimerAltList`** auxquelles un temps d'attente maximal est associé, puis **`ProcSkipAlt`** et **`ProcSkipAltList`** pour n'effectuer qu'un simple test des canaux sans rester bloqué.

Par contre la portée de toutes ces fonctions est restreinte au processeur où elles sont employées. De ce fait un processus nouvellement créé s'exécutera sur le processeur où l'appel est effectué. Il en est de même pour les canaux de communication qui ne peuvent relier que deux processus qui s'exécutent sur le même noeud. Toutefois, les liens de communication d'un noeud sont accessibles au travers de canaux spéciaux, ce qui rend l'échange de messages entre noeuds voisins possible; mais c'est à l'utilisateur de programmer le routage des messages pour réaliser des communications entre toute paire de processeurs.

De nouvelles versions du compilateur, qui «supportent» notamment les derniers produits de la gamme des transputers: le **T9000** et le routeur **C104** [MTW93], pallient en partie à ces problèmes en automatisant le routage. Pour des réseaux de transputers **T800** ou **T400**, l'utilisateur dispose de canaux virtuels pour relier des processus indépendamment des processeurs sur lesquels ils s'exécutent; dans ce cas un routeur de type **VCR** (Virtual Channel Router[1]) [DHN90, DHN91] est mis en oeuvre. Pour des réseaux de T9000 ce sont les canaux internes du T9000 et le routage par intervalles des C104 qui assurent cette tâche. Il ne reste plus alors qu'à configurer les routeurs en leur transmettant les tables de routage.

Ces restrictions s'expliquent entre autres par l'absence d'un véritable support système. Pour cette même raison, et parce qu'il reprend les concepts de CSP, le C Inmos n'offre pas de fonction de diffusion.

A.2 Un langage concurrent: Ada

Ada[DoD82] est né de la volonté du *Département Américain de la Défense* d'avoir un langage susceptible de remplacer ceux, et ils étaient nombreux, déjà utilisés dans ses divers services. Sa conception et sa normalisation dura près de dix ans de 1974 à 1983. De conception rigoureuse il est l'héritier de langages structurés tels que Pascal et MODULA, auxquels il incorpore des possibilités nouvelles issues de la programmation orientée objet comme la surcharge et la généricité. Ada n'est donc pas à vrai dire un langage de programmation parallèle; il s'agit plutôt d'un langage concurrent où l'exécution de tâches parallèles ou pseudo-parallèles fait partie intégrante du langage.

Le modèle de parallélisme sur lequel se base Ada est celui des processus communicants de CSP. Ainsi est-il possible de déclarer des types **`task`** (tâche) auxquels pourront correspondre plusieurs instanciations, i. e. plusieurs tâches. Les communications et synchronisations se font par échanges de messages lors de rendez-vous. De plus comme les règles d'accès aux fonctions, procédures, variables globales et aux paquetages s'appliquent identiquement aux tâches, celles-ci peuvent également communiquer par partage de mémoire.

1. Routeur à Canaux Virtuels.

La déclaration d'une tâche, à l'image de ce qu'est la classe pour un objet, commence par la spécification du modèle de tâche auquel elle appartient, c'est-à-dire la description des points d'entrée qui serviront aux communications, et dont l'utilisation et la syntaxe sont similaires à celles des procédures. Si par exemple il s'agit de gérer un journal selon la politique des lecteurs/rédacteurs, nous pouvons avoir:

```
-- on suppose connus le type ARTICLE et TAILLE_JOURNAL
task type JOURNAL is
    entry INIT;
    entry LIRE( I:in INTEGER range 1..TAILLE_JOURNAL;
                P:out ARTICLE);
        -- un tableau d'entrées
    entry ECRIRE(   I:in INTEGER range 1..TAILLE_JOURNAL;
                    P:in ARTICLE);
    entry FIN_LECTURE;
    entry DEBUT_ECRITURE;
end JOURNAL;
```

Le type `task` ainsi spécifié sert alors à la déclaration de tâches qui seront créées statiquement au lancement du programme, mais aussi pour des objets de type `access` qui permettront la création dynamique de tâches grâce à l'opérateur `new`:

```
LE_MONDE: JOURNAL; -- journal créé statiquement
type FANZINE is access JOURNAL;
LAIUS, ES2I_RIDER: FANZINE; -- journaux créés dynamiquement
```

Il faut aussi décrire le «corps» de la tâche, c'est-à-dire l'algorithme principal qui régit l'exécution des tâches:

```
task body JOURNAL is
    -- partie déclarative
    type T_JOURNAL is array (1..TAILLE_JOURNAL) of ARTICLE;
    RECEUIL: T_JOURNAL;

    NB_LECTEURS: INTEGER;
    REDACTEUR: BOOLEAN;
    -- etc.
begin
    -- corps du gestionnaire de journal
end JOURNAL;
```

Le mécanisme de communication de base est l'échange de messages par rendez-vous. Pour cela il suffit d'effectuer un appel à l'un des points d'entrée de la tâche, de la même manière que l'on appelle une procédure; à titre d'exemple:

```
LE_MONDE.INIT;
LE_MONDE.ECRIRE(2,PAPIER); -- avec PAPIER de type ARTICLE
LAIUS.LIRE(1,LA_UNE); -- de même, LA_UNE de type ARTICLE
ES2I_RIDER.LIRE(10,POTINS); -- idem pour POTINS
```

L'exécution d'une telle instruction est mise en attente jusqu'à ce que le rendez-vous avec la tâche puisse s'établir, soit lorsque la tâche appelée accepte le rendez-vous, ou immédiatement si celle-ci est déjà en attente sur le point d'entrée. Pour la tâche appelée l'acceptation d'un rendez-vous est un appel explicite où doivent être précisés le point d'entrée avec ses paramètres formels, ainsi qu'un bloc d'instructions qui seront exécutées durant le rendez-vous. Le rendez-vous prendra fin lorsque l'exécution de ce bloc d'instructions sera terminée. Dans notre exemple, l'initialisation d'un journal se fait en début d'exécution lors d'un rendez-vous:

```
task body JOURNAL is
    -- partie déclarative, comme ci-dessus
begin
    accept INIT do
        -- on suppose défini ARTICLE_VIDE
        RECEUIL := (1..TAILLE_JOURNAL => ARTICLE_VIDE);
        NB_LECTEURS := 0;
        REDACTEUR := FALSE;
    end INIT;
    -- sessions de lectures/rédactions
end JOURNAL;
```

L'utilisation de cette construction est strictement limitée au corps de la tâche (et aux blocs locaux à ce corps); il n'est donc pas permis d'en faire usage dans les sous-programmes, paquetages ou tâches différentes. Par conséquent, non seulement une tâche ne peut accepter de rendez-vous que sur ses propres entrées, mais il y a de surcroît exclusion mutuelle sur celles-ci; ce qui interdit d'ailleurs le parallélisme des rendez-vous.

En revanche Ada permet d'exprimer l'attente conditionnelle, l'attente multiple, l'attente limitée dans le temps et le non-déterminisme, en d'autres termes le langage implémente les commandes gardées [Dijk75]. Pour un fonctionnement sans contrôle, une session d'accès au journal pourrait être la suivante:

```
select
    accept LIRE(I:in INTEGER range 1..TAILLE_JOURNAL;
                P:out ARTICLE) do
        P := RECUEIL(I);
    end LIRE;
or
    accept ECRIRE(I:in INTEGER range 1..TAILLE_JOURNAL;
                  P:in ARTICLE) do
        RECEUIL(I) := P;
    end ECRIRE;
end select;
```

Ici la tâche se met en attente sur ses deux entrées `LIRE` et `ECRIRE`, lorsque l'une d'elles devient prête pour un rendez-vous celui-ci s'établit et le `select` se comporte comme un simple `accept` sur cette entrée. En cas de concurrence sur les entrées, un choix non déterministe est opéré.

Pour imposer des conditions à l'attente sur certaines entrées, comme cela est possible avec les commandes gardées, chaque branche peut être soumise à des contraintes exprimées par une expression booléenne. C'est par exemple ce qui est fait dans le modèle des lecteurs/rédacteurs:

```
select
    when not REDACTEUR =>
        accept LIRE(I:in INTEGER range 1..TAILLE_JOURNAL;
                    P: out ARTICLE) do
            P := RECEUIL(I);
            NB_LECTEURS := NB_LECTEURS + 1;
        end LIRE;
    or when NB_LECTEURS > 0 =>
        accept FIN_LECTURE do
            NB_LECTEURS := NB_LECTEURS - 1;
        end FIN_LECTURE;

    or when NB_LECTEURS = 0 and not REDACTEUR =>
        accept DEBUT_ECRITURE do
            REDACTEUR := TRUE;
            end DEBUT_ECRITURE;

    or when REDACTEUR =>
        accept ECRIRE(I:in INTEGER range 1..TAILLE_JOURNAL;
                      P:in ARTICLE) do
            RECEUIL(I) := P;
            REDACTEUR := FALSE;
        end ECRIRE;

end select;
```

Plutôt que seulement des acceptations de rendez-vous, les options du `select` peuvent aussi être temporisées (instruction `delay`), et dans ce cas l'attente sur les diverses entrées est limitée au minimum des délais spécifiés: lorsque durant ce délai aucun rendez-vous n'a été effectué, la branche du `select` associée au délai est exécutée et le `select` prend fin. Il est également possible de limiter l'attente à la durée de vie de l'environnement de la tâche (option `terminate`), et garantir ainsi la terminaison correcte des tâches. Enfin, cette construction admet une option `else` qui est activée lorsqu'aucune autre branche ne peut l'être immédiatement; le `select` n'est alors plus bloquant.

Du côté appelant l'utilisation du `select` est restreint à la demande de rendez-vous immédiat (mode non bloquant) et à l'attente limitée dans le temps. Par exemple:

```
select
    LE_MONDE.INIT;
        -- l'initialisation du journal s'est faite
    else
        -- le journal est déjà initialisé
end select;
select
    LE_MONDE.LIRE(1,LA_UNE);
        -- lecture de LA_UNE;
        LE_MONDE.FIN_LECTURE;
    or delay 60.0
        -- journal inaccessible après 1 min. d'attente
end select;
```

Malgré un modèle basé sur l'échange de messages, Ada reste un langage difficile à implanter sur les machines parallèles sans mémoire commune. En effet le partage d'informations, sous-programmes et données, qu'il offre rend délicate son implémentation sur ces machines, car peu d'entre elles disposent d'un mécanisme de partage virtuel de la mémoire, aussi bien logiciel que matériel. Comme nous l'avons vu avec le système Amoeba, un tel mécanisme peut être offert à partir de la diffusion. Elle pourrait également permettre de réaliser l'exclusion mutuelle inhérente à ce langage. Hormis ces possibles utilisations, la diffusion n'est pas directement requise par Ada.

A.3 Un langage parallèle: Occam

Si Ada reprend l'essentiel des concepts de CSP pour gérer le parallélisme, Occam [Inmos82, Inmos84, PouMay88] est lui une véritable implémentation de ce modèle. Conçu pour les architectures à échange de messages, il est fortement lié au transputer pour lequel il a été à l'origine développé. Il est d'ailleurs fréquent, même si cela est abusif, d'entendre dire qu'il en est le langage assembleur. Toutefois il est disponible pour d'autres architectures et types de processeurs.

Trois concepts fondamentaux régissent la programmation en Occam:

- toute opération, de la plus simple à la plus complexe, de l'instruction élémentaire au programme tout entier, est considérée comme un processus;
- un programme est une composition de processus;
- les processus communiquent et se synchronisent par échanges de messages sur des **canaux** synchrones unidirectionnels.

Contrairement à bien d'autres langages l'exécution séquentielle d'instructions n'est pas ici implicite; elle est remplacée par la composition séquentielle de processus. De même, les instructions conditionnelles et répétitives sont considérées comme des processus. Le programme séquentiel suivant, qui calcule pour un entier donné une suite d'entiers qui aboutit à une puissance de 2 puis à 1, illustre bien cette différence:

```
SEQ
  nb_iter, nb_pair, nb_impair := 0, 0, 0

  WHILE n <> 1
    SEQ
      nb_iter := nb_iter + 1
      IF
        (n REM 2) = 0
          SEQ -- exécuté si n est pair
            nb_pair := nb_pair + 1
            n := n / 2
        TRUE
          SEQ -- exécuté sinon
            nb_impair := nb_impair + 1
            n := 3 * n + 1
```

Dans la composition parallèle, dénotée par le processus **PAR**, de laquelle est issue la sémantique des primitives `ProcPar` et `ProcParList` du C Inmos, les processus démarrent simultanément et se synchronisent à la fin de leurs exécutions. L'exemple suivant est équivalent, quant au résultat, au précédent:

```
WHILE n <> 1
    PAR
      nb_iter := nb_iter + 1
      IF
        (n REM 2) = 0
          PAR -- exécuté si n est pair
            nb_pair := nb_pair + 1
            n := n / 2
        TRUE
          PAR -- exécuté sinon
            nb_impair := nb_impair + 1
            n := 3 * n + 1
```

A l'aide des constructeurs SEQ[1] et PAR il est possible d'effectuer, respectivement, des itérations séquentielles ou de répliquer un processus; ex.:

```
SEQ i = 1 FOR SIZE tab
  tab[i] := 0

PAR i = 1 FOR 10
  tab[i] := 0
```

Selon la même philosophie les procédures et fonctions sont des processus, qui lorsqu'elles sont invoquées interrompent durant leur exécution le processus appelant.

Si les processus séquentiels d'un SEQ partage l'accès aux variables, cela n'est pas permis entre les processus d'un PAR; une variable ne peut être utilisée que par l'un d'entre eux et seulement celui-ci. En effet, la communication en Occam se fait essentiellement par des échanges de messages à travers des canaux qui, parce qu'il y a rendez-vous entre les processus communicants, permettent de plus de synchroniser les processus.

Un canal Occam relie deux processus, toujours les mêmes, l'un en émission et l'autre en réception. Non seulement il n'est pas autorisé d'effectuer des communications en sens inverse (du récepteur vers l'émetteur), mais il est également interdit d'utiliser un canal pour des processus autres que le récepteur et l'émetteur. De plus un canal est associé à un «protocole» qui représente le type d'informations qu'il véhicule. Les opérateurs définis dans CSP: **"!"** pour l'émission et **"?"** pour la réception, ont été repris pour les canaux Occam. Le programme ci-après illustre sur un cas très simple le principe de fonctionnement des canaux:

```
CHAN OF REAL32 Canal_pi :
REAL32 pi:
PAR
  Canal_pi ! 3.14159
  Canal_pi ? pi
```

Pour des structures de données moins simples que les types de base, l'utilisateur peut créer des protocoles complexes. Ce peut être un protocole qui correspond simplement à un type de base du compilateur (un entier par exemple), un protocole séquentiel composé de plusieurs champs et/ou des tableaux de tailles fixes, un protocole à partie variable, ou même un protocole pour les tableaux de tailles variables. Les déclarations ci-dessous montrent les possibilités offertes:

1. le processus conditionnel IF peut de même être exprimé sous une forme réplicative.

```
PROTOCOL Prot_simple IS INT; INT; INT :
PROTOCOL Prot_tableau_fixe IS [100]REAL64 :
PROTOCOL Prot_tableau_variable INT::[]BYTE :
    -- le premier type est associé à la taille du tableau
    -- qui devra être transmise lors de la communication
PROTOCOL Prot_a_variante
  CASE
    cas_1; INT64
    cas_2; INT32; BYTE; BOOL
:
```

Dans le cas de tableaux à taille variable, l'émetteur doit communiquer la taille du tableau (ou de la partie du tableau) qu'il transmet. Outre l'obligation pour le récepteur de recevoir cette information dans une variable, il faut de surcroît s'assurer que le tableau prévu en réception est suffisant pour stocker ce qui est envoyé.

Si en émission il faut simplement se placer dans le cas voulu, en réception l'utilisation des protocoles à partie variable est un peu plus compliquée. Si l'on sait quelles données vont être envoyées il suffit juste de préciser le cas correspondant. Autrement l'appel doit prendre en compte tous les cas. En fait la réception s'apparente alors à une opération de sélection, où l'utilisateur associe un processus à chaque cas. Toutefois il ne faut pas confondre ce type de protocole avec le processus de sélection **CASE**.

Occam offre également les commandes gardées, mais l'implémentation, qui a dans ce langage le nom d'alternative, est ici plus proche de la sémantique de CSP. Par ailleurs les différents cas d'un protocole peuvent être pris en compte dans une alternative. Enfin tout comme pour les processus PAR et SEQ, l'alternative (**ALT**) peut être répliquée. Dans l'exemple suivant il s'agit simplement de réaliser un tampon borné, géré comme une file d'attente:

```
PAR
  VAL INT MAX_CLIENT IS 100 :
  VAL INT TAILLE_MAX IS 1024 :
  [MAX_CLIENT]CHAN OF BYTE Ecrire_tampon :
  [MAX_CLIENT]CHAN OF BOOL Lire_tampon :
  [MAX_CLIENT]CHAN OF BYTE Réponse_tampon :

  -- les clients

  SEQ -- le processus de gestion du tampon
    [TAILLE_MAX]BYTE tampon :
    INT tête, nb_item :
    BOOL Requète_lecture :
    BYTE item :

    tête, nb_item := 0, 0
    WHILE TRUE
      ALT i = 0 FOR MAX_CLIENT
        (nb_item < TAILLE_MAX) & Ecrire_tampon[i] ? item
          SEQ
            tampon[(tête + nb_item) REM TAILLE_MAX] := item
            nb_item := nb_item + 1
```

```
(nb_item > 0) & Lire_tampon ? Requète_lecture
  SEQ
    Réponse_tampon[i] ! tampon[tête]
    tête := (tête + 1) REM TAILLE_MAX
    nb_item := nb_item - 1
```

Occam dispose également d'opérateurs pour manipuler le temps, à partir desquels l'attente dans une alternative peut être limitée dans le temps; ce qui en fait un langage apte à prendre en compte des contraintes temporelles de type temps critique.

Les premières réalisations d'Occam pour des architectures parallèles sans mémoire commune disposaient de primitives de placement de processus, cependant seules les communications entre processeurs voisins étaient possibles, comme avec le C Inmos précédemment décrit. Ceci est devenu disponible avec l'évolution des processeurs et routeurs, notamment avec les T9000 et C104 d'Inmos qui offrent la communication entre processeurs non voisins, et permettent également de définir des canaux, en nombre limité certes, entre des processus distants. Les dernières versions du langage [Bar92] introduisent, entre autres, un nouveau type de canaux: les canaux d'appel de procédures à distance.

Occam est un langage qui ne possède aucun opérateur de diffusion. Tous deux sont en effet construits sur le modèle de CSP qui ne définit qu'un seul mode de communication: l'échange de messages point-à-point synchrone.

A.4 Comparaison

Cette présentation des langages parallèles ainsi que celles des langages Linda et Esterel, avec également celle des environnements de développement et d'exécution du chapitre 2, nous montrent combien peuvent être variés les outils de programmation parallèle. Leur comparaison, présentée dans le tableau A.1 ci-après, nous permet de remarquer que les langages diffèrent des environnements de développement dans l'expression du parallélisme et dans les possibilités de communications. Alors que les langages offrent le support sémantique de composition parallèle, l'ensemble des objets de communication est restreint. A l'inverse, pour les environnements de développement, les primitives d'expression du parallélisme sont très réduites face à la grande diversité des fonctions de communication. Cette divergence est très nette avec MPI qui, comme résultat de l'évolution de ses prédécesseurs, ne propose aucune fonction de gestion des processus au profit d'une bibliothèque très vaste de primitives d'échange de messages.

	Composition parallèle		Groupe de processus	Placement et alignement des données	Placement des calculs	Communication point-à-point		Gardes	Communications globales			Gestion mémoire partagée
	bloquant	¬ bloquant				bloquant	¬ bloquant		diffusion	barrière de synchro	autres	
HPF	`FORALL`			`BLOCK` `CYCLIC` `BLOCK_CYCLIC`	par rapport aux données							automatique
MODULA-3	`FORALL`	`NOSYNC`		`LOCAL` `SPREAD` `CYCLIC`	par rapport aux données							automatique
CC++	`par` `parfor`	`spawn`										`sync` `atomic`
C Inmos	`ProcPar`	`ProcRun`			statique	canaux		`ProcAlt`				
Ada		modèle de tâches				rendez-vous		`select`				
Occam	`PAR`				statique	canaux		`ALT`				
Esterel	`||`							`await`	signaux	signaux		
PVM		`spawn` `mytid`	`joingroup` `lvgroup`		manuel	`recv`	`send` `nrecv` `sendsig`		`mcast` `bcast` `notify`	`barrier`		
p4					statique	`recv` `sendr`	`send` `messages-available`		`broad-cast`	`global_ barrier`	calculs	`shmalloc` `shfree` moniteurs
MPI			groupe ensembliste		topologies de processus	`standard` «ready» `synchrone`	`standard` «ready» `synchrone`		`BCAST`	`BARRIER`	très complet	

Tableau A.1 : comparaison des langages et environnements parallèles.

Annexe B : Systèmes d'Exploitation Distribués et Parallèles

L'émergence des architectures parallèles ces dernières années est, au moins sur ce point, comparable à celle des réseaux locaux d'ordinateurs du début des années 80; ce sont les mêmes types de problèmes qui furent posés pour proposer une structure globale de calcul et de stockage de données à partir d'un ensemble d'ordinateurs interconnectés par un réseau local. Monolithiques et de taille difficilement maîtrisable - 3 à 5 millions de lignes de code assembleur pour VMS sur un VAX - les systèmes d'exploitation classiques se sont avérés inaptes à une telle évolution. Qu'ils soient appelés *systèmes distribués* ou *systèmes répartis*, toutes les solutions proposées ont adopté le même principe: fonder tous les services et programmes systèmes sur un micro-noyau composé seulement des fonctions les plus essentielles [Gien90].

B.1 Chorus

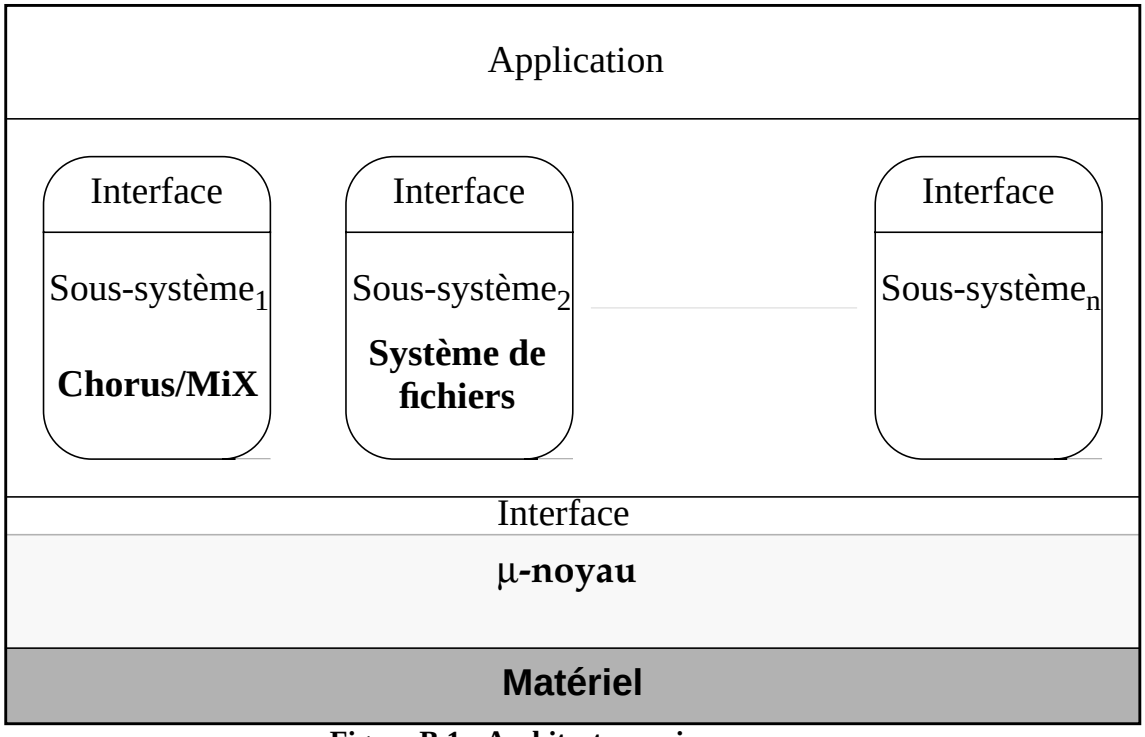


Figure B.1 : Architecture micro-noyau.

L'architecture de Chorus [RozAbro90, HMA90, Guil90], représentative de ce qui est fait dans tout système réparti, se construit autour d'un micro-noyau à partir duquel divers sous-systèmes sont définis et proposent aux applications les fonctionnalités habituelles. La figure B.1 montre cette structure et les diverses couches qui la caractérisent. Les sous-systèmes offrent des servi-

ces séparés et coopérants; ils cohabitent et interagissent les uns avec les autres selon une structure modulaire. Les services rendus peuvent être locaux pour la gestion de ressources locales, ou globaux. Dans le cas de services globaux le noyau gère la distribution et la coopération entre les sites. Un sous-système se compose d'un ou plusieurs serveurs et d'une bibliothèque qui sert d'interface.

Chorus/MiX [AHLR90, AGP91] qui reproduit complètement l'interface Unix standard est l'un des sous-systèmes les plus représentatifs de cette réalisation. Il s'appuie entre autres sur un serveur de fichiers en tout point compatible avec celui d'Unix, mais pour un environnement distribué.

Avec cette philosophie de construction le noyau se trouve réduit et sa structure adaptée au petit nombre de fonctions dont il a la charge: l'**Exécutif Temps Réel** qui réalise l'ordonnancement des processus et qui est capable de prendre en compte des contraintes temporelles, le **Superviseur** qui contrôle et oriente les interruptions logicielles et matérielles, le gestionnaire de communication (**IPC Manager**), et le gestionnaire de mémoire virtuelle (**PVM**) [ARS89, ARG89] (voir figure B.2). Dans cette structure seul le superviseur et une partie de PVM sont dépendants de la machine; les autres composants sont entièrement portables.

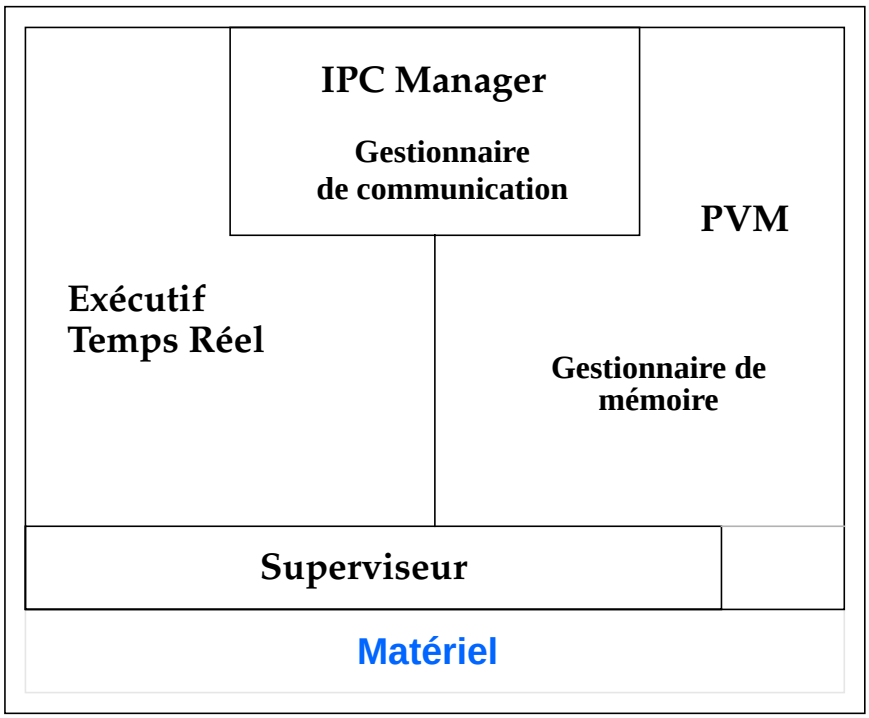


Figure B.2 : structure du micro-noyau de Chorus.

Le modèle d'exécution de Chorus se décompose en deux niveaux de processus: les ***acteurs***[1] et les ***threads***. L'acteur est l'unité d'allocation de ressources; il regroupe un ensemble de threads dans un même espace d'adressage protégé. Il existe trois types d'acteurs selon le mode d'exécution des threads: *utilisateur* sans droit particulier, *système* qui donne accès aux opérations sensibles du noyau, et *superviseur* qui permet en plus l'utilisation des instructions privilégiées du processeur. Un acteur est lié à un site fixe, ce qui impose à ses threads de ne pas pouvoir s'exécuter sur d'autres sites.

1. actors dans la terminologie anglaise de Chorus.

Composant d'un seul et unique acteur, le thread est l'unité d'exécution et d'ordonnancement. Il s'agit d'un processus léger, ou plus exactement d'un flot de contrôle. Au sein d'un même acteur les threads sont en concurrence les uns avec les autres pour l'accès au processeur - une exécution parallèle est toutefois possible dans le cas de machines multi-processeurs - selon une politique d'ordonnancement adaptée à l'application: temps partagé, avec ou sans préemption, à priorités fixes ou évolutives. Du fait de sa structure l'acteur fournit à ses threads une première forme de communication: la mémoire partagée. Cependant Chorus n'offre pas d'outils de synchronisation autres que les ports ou que ceux intégrés au processeur.

Le ***port*** est donc le seul objet de communication géré par le noyau. Il représente une zone ordonnée de mémoire pour le stockage de messages non consommés. Attaché à un seul acteur à la fois, seuls les threads de cet acteur peuvent y lire les messages. Cette relation d'appartenance est dynamique, si bien qu'un port peut *migrer* d'un acteur à un autre; mais ce changement de propriétaire ne concernera que les messages non consommés, les anciens messages n'étant plus accessibles. Les communications à travers un port se font d'un émetteur vers un récepteur, où l'émetteur est quelconque et le récepteur uniquement un thread de l'acteur qui s'est approprié le port. La communication peut se faire en mode asynchrone, dans ce cas l'émetteur ne reste bloqué que jusqu'à ce que son message soit copié dans le port, sans garantie de sa réception. Elle peut aussi se faire en mode RPC[1], et dans ce cas l'émetteur attend d'être averti de la réception de son message, d'une panne ou de tout problème de communication; le fonctionnement est alors du type *Client-Serveur.*

Pour assurer des services distribués, rendre le système tolérant aux pannes ou offrir des possibilités de diffusion, Chorus gère également des groupes de ports. Ensembles de ports, ces groupes sont manipulés dynamiquement par insertion et suppression de ports. Un port peut appartenir de surcroît à plusieurs groupes simultanément, ce qui rend ces ensembles plus souples d'utilisation. L'envoi d'un message à un groupe peut se faire vers tous les membres, vers un membre quelconque, vers les membres d'un site précis, ou vers les membres du même site que celui de l'expéditeur. La figure B.3 schématise la conception d'une application comme ensemble de serveurs distribués et coopérants à partir de ces outils de communication.

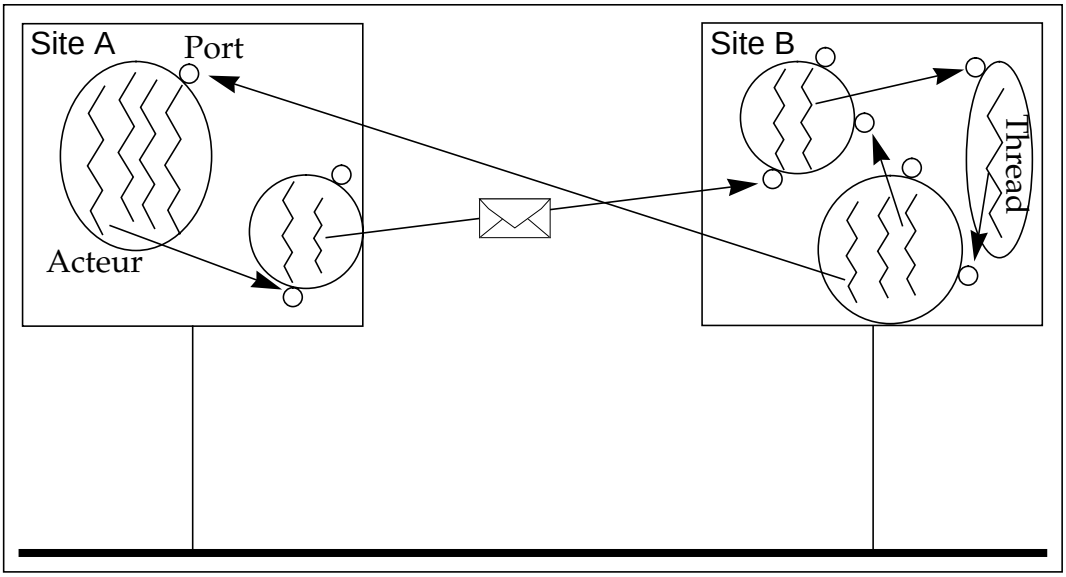


Figure B.3 : structure d'une application distribuée sous Chorus.

1. Remote Procedure Call ou appel de procédure à distance.

La désignation de tous ces objets se fait grâce à trois types d'identificateurs: *uniques*, *locaux*, et *capacités*. Un identificateur unique (**UI**) est commun et global à tous les sites; le noyau assure la transparence de localisation de l'objet associé. Les UI sont utilisés pour les acteurs, les ports et les segments de mémoire, i. e. pour tous les objets et ressources globaux manipulés par le noyau.

Un identificateur local n'appartient qu'au contexte d'un objet, par exemple pour désigner les threads d'un acteur. Il n'a, en général, aucun sens hors de son site de création.

Pour étendre le service de désignation à des objets externes au noyau, Chorus gère également des capacités. Ce sont des noms globaux qui associent l'UI du serveur de l'objet et une clé donnée par le serveur, clé qui contient l'identification locale de l'objet ainsi que des droits d'accès. Une capacité peut donc être associée à n'importe quel type d'objet (acteur, port, thread, etc.).

Essentielle aux applications temps réel ou au contrôle de périphériques, la gestion des exceptions et des événements matériels est une fonction programmable du système. L'accès direct aux événements d'entrées/sorties et aux exceptions matérielles se fait par l'affectation du traitement d'une interruption à un thread d'un acteur en mode superviseur. Le déclenchement de l'interruption provoquera l'appel prioritaire du thread de traitement de l'interruption. Il est de plus possible d'associer, à l'intérieur de tout acteur, une exception à un port, et ainsi d'effectuer automatiquement l'envoi d'un message plutôt que d'exécuter une routine de traitement.

Dernier composant du noyau, le gestionnaire de mémoire virtuelle considère deux entités: les *segments* et les *régions*. Une région est une partie de l'espace d'adressage d'un acteur auquel peut être attaché, avec des droits d'accès, une portion de segment. Lors de chaque référence à une région le noyau vérifie seulement que l'opération respecte ces droits; aucun autre mécanisme n'est mis en oeuvre à ce niveau pour ne pas dégrader les performances. Le segment est l'unité d'information que manipule le système. Global à tout le système, il est en général implémenté à partir des mémoires secondaires et géré par des acteurs systèmes, les **«mappers**», qui assurent la politique d'accès, la protection, le contrôle de cohérence, et surtout réalisent l'attachement des segments aux régions. Le segment est avant tout un objet partageable entre plusieurs acteurs indépendamment du site où ils s'exécutent; la cohérence est donc une fonction essentielle des mappers.

La définition d'un sous-système se fait comme un ensemble d'acteurs qui dote le système d'une interface de programmation et d'exploitation des ressources, ainsi que les abstractions classiques d'un système: processus, encapsulation des données, etc. La figure B.4 nous montre les possibilités offertes à ce niveau, et notamment celles qui consistent à définir des appels système, ou «**traps**».

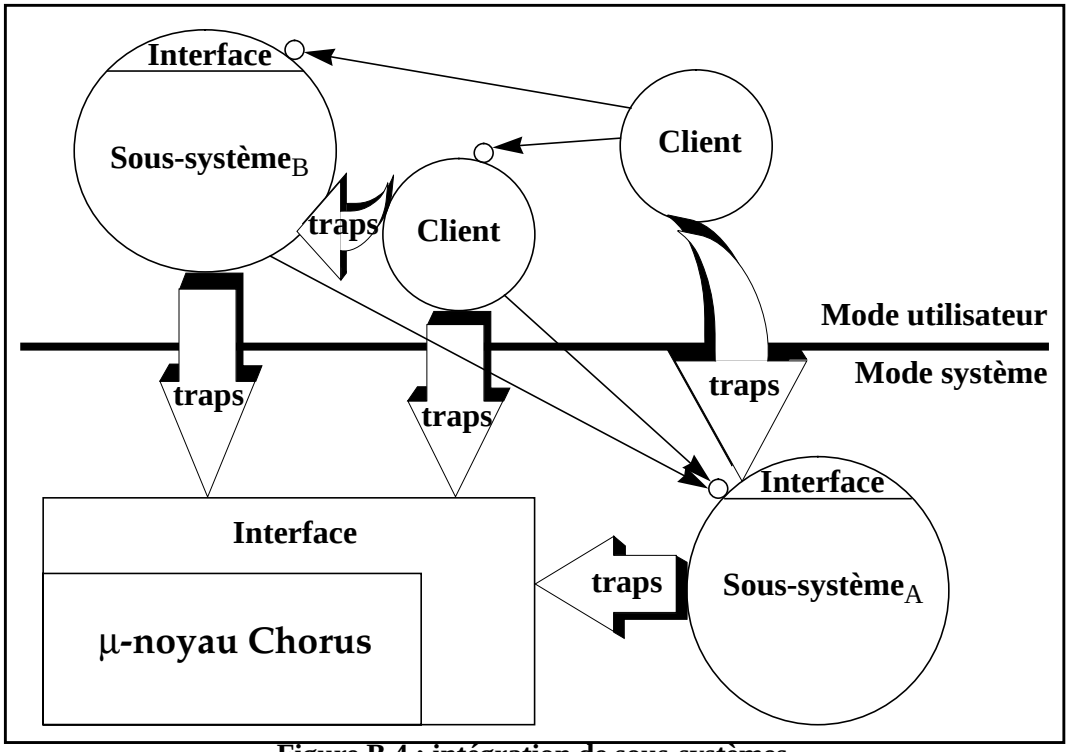


Figure B.4 : intégration de sous-systèmes.

B.2 Mach

Les similitudes entre Chorus et Mach [ABG86, FGB91, RJO89, RBF89] sont nombreuses; en effet ces deux systèmes ne diffèrent vraiment que par rapport aux choix de réalisation qu'ils ont opérés. Le rôle du micro-noyau est ici aussi de gérer et d'ordonnancer les processus, d'offrir un mécanisme de base pour la communication et de fournir une mémoire virtuelle indépendante de l'architecture matérielle.

Le modèle de processus de Mach est lui aussi constitué de deux niveaux de processus: les ***threads*** qui sont les flots de contrôle, et les ***tâches*** (*tasks*) qui, comme les acteurs de Chorus, regroupent dans un même espace d'adressage un ensemble de threads. De même le modèle de communication repose sur l'échange de messages et sur un unique objet: le ***port***. Structure de données capable de stocker des messages, le port n'appartient ici qu'à une et une seule tâche. La protection d'un port est assurée par le noyau qui lui associe une ***capacité***. Pour accéder à un port, en réception comme en émission, un processus doit identifier le port avec sa capacité, ce qui n'est possible que lorsque cette capacité a été transmise au processus. En plus de la sécurité et de la fiabilité de ce dispositif, le noyau garantit la transparence de localisation et le typage des données échangées. Les ports permettent aux tâches de coopérer selon un modèle *Client-Serveur* asynchrone, seul mode de communication offert par le noyau. Il n'existe pas, comme dans Chorus, d'autre mode de fonctionnement, ni de notion de groupes de ports.

Les ports servent également à la gestion de la mémoire virtuelle. L'association de ces deux mécanismes de communication assurent le transfert de gros volumes de données à un coût moindre. En effet les objets mémoires sont, à l'intérieur de Mach, représentés par des ports; et le système assure les transferts lors de défauts de pages ou de copies en écriture[1].

1. copy-on-write.

Les mécanismes de déroutement d'exceptions et d'interruptions sont également disponibles, mais dans Mach cela est réalisé par héritage de bibliothèques de fonctions. Chargées dans l'espace d'adressage des programmes, les bibliothèques sont héritées systématiquement, et de manière transparente, à chaque création de processus. C'est dans ces bibliothèques que sont redirigés les appels systèmes, ce qui en plus permet d'assurer la compatibilité binaire avec des environnements différents de ceux de Mach, d'offrir un support pour divers environnements systèmes (Unix par exemple), de développer des outils adaptés pour le déverminage et l'observation du comportement des applications, ou plus simplement de traduire un appel système en une communication sur un port.

Enfin, Mach est fortement orienté objet. Il est doté de toutes les fonctionnalités courantes de la programmation par objets, fonctionnalités intégrées aux mécanismes de communication:

- la spécification dynamique de classes et de méthodes;
- une hiérarchie en classes et super-classes;
- l'héritage multiple grâce à la délégation;
- la délégation distante automatique;
- etc.

Le serveur Unix offert par Mach est l'une des principales raisons de son succès. Non seulement cela prouve la souplesse de ce type d'architectures de systèmes, et son aptitude à s'adapter à tout type de contraintes, mais de surcroît il s'avère plus performant que les implantations classiques d'Unix [GDFR90].

Ce serveur est simplement une tâche composée d'une douzaine de threads affectés aux diverses fonctions: contrôle des périphériques, gestion des i-nodes, et pour la plupart le traitement des appels systèmes effectués par les applications. Comme le montre la figure B.5 l'accès à ce serveur est transparent et est pris en charge par une bibliothèque liée à tout programme.

A travers les fonctions inclues dans la bibliothèque, une application Unix communique avec le serveur à partir des outils de communication de Mach, c'est-à-dire les ports. Ainsi un appel système est transformé en l'envoi d'un message au serveur, message qui contient l'opération à effectuer ainsi que ses paramètres; puis ce message est pris en compte par l'un des threads de service qui assure alors le bon déroulement de l'opération. Cependant tout appel système ne se traduit pas nécessairement par un message. Par exemple, pour l'accès aux fichiers il est préférable d'utiliser le plus possible les possibilités de la mémoire virtuelle. Un fichier est en effet un objet mémoire placé dans l'espace d'adressage du programme, où seuls les défauts de page font intervenir le serveur Unix (cf. figure B.6).

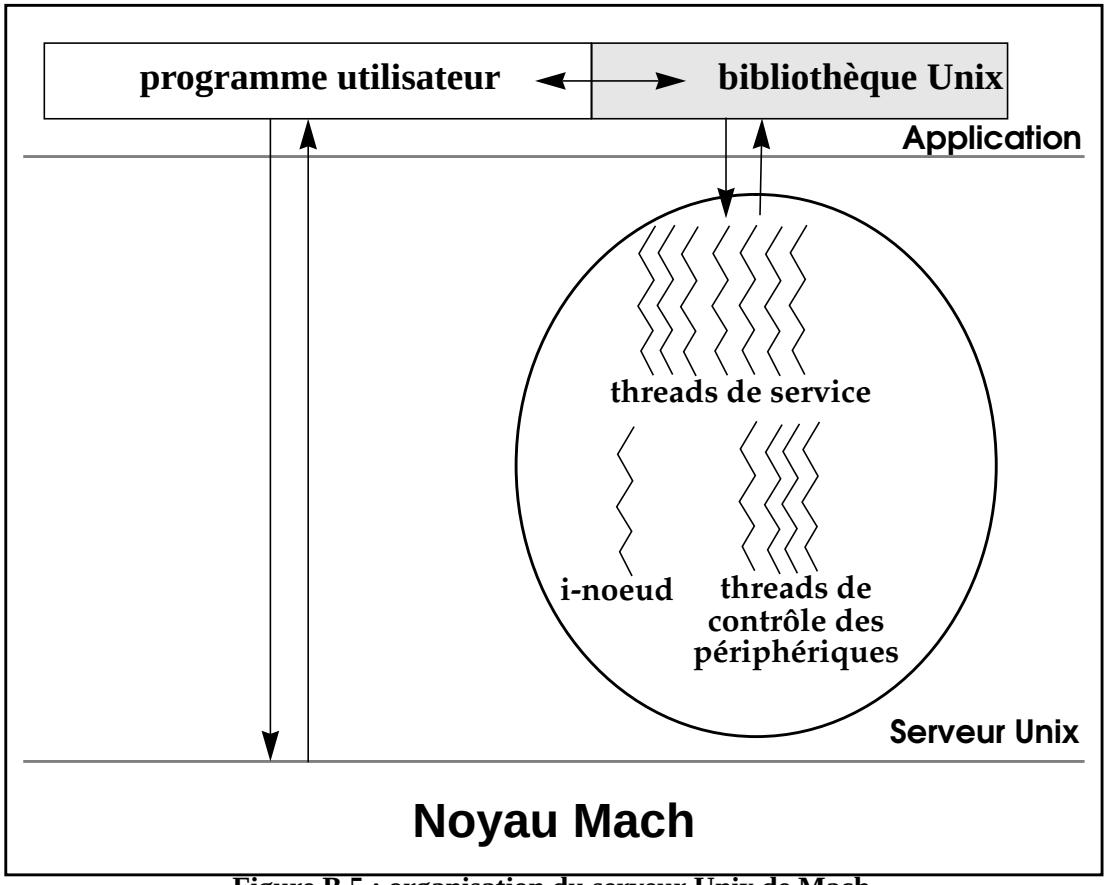


Figure B.5 : organisation du serveur Unix de Mach.

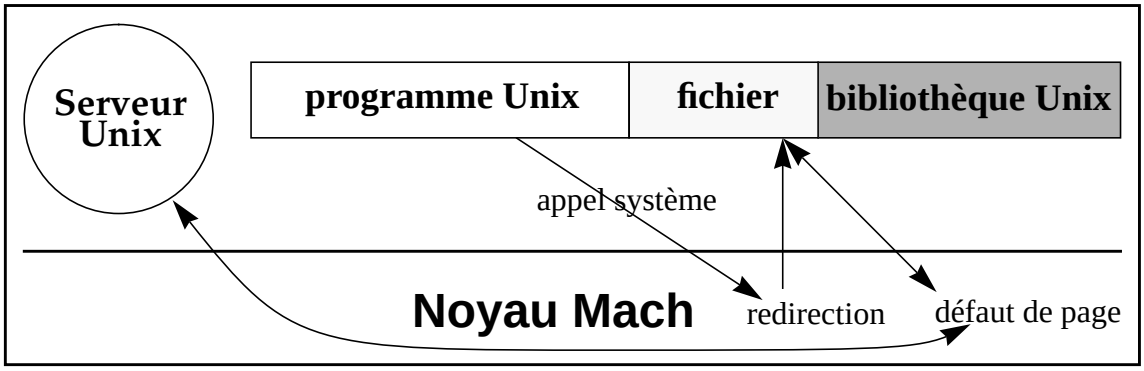


Figure B.6 : accès local à un fichier sous Mach.

B.3 Trollius

L'objectif principal de Trollius [BDV94, Burns88, BRDM90] est simple: gérer les fonctionnalités matérielles d'une machine parallèle. Il offre une vision uniforme de ce type de machines: une architecture à mémoire distribuée. Ainsi, seul le modèle à échange de messages est considéré. Dans cet esprit Trollius ne se limite pas à celui d'une machine unique, mais offre l'accès à un ensemble de machines parallèles par l'intermédiaire d'ordinateurs frontaux, connectés à un réseau local comme le montre la figure B.7.

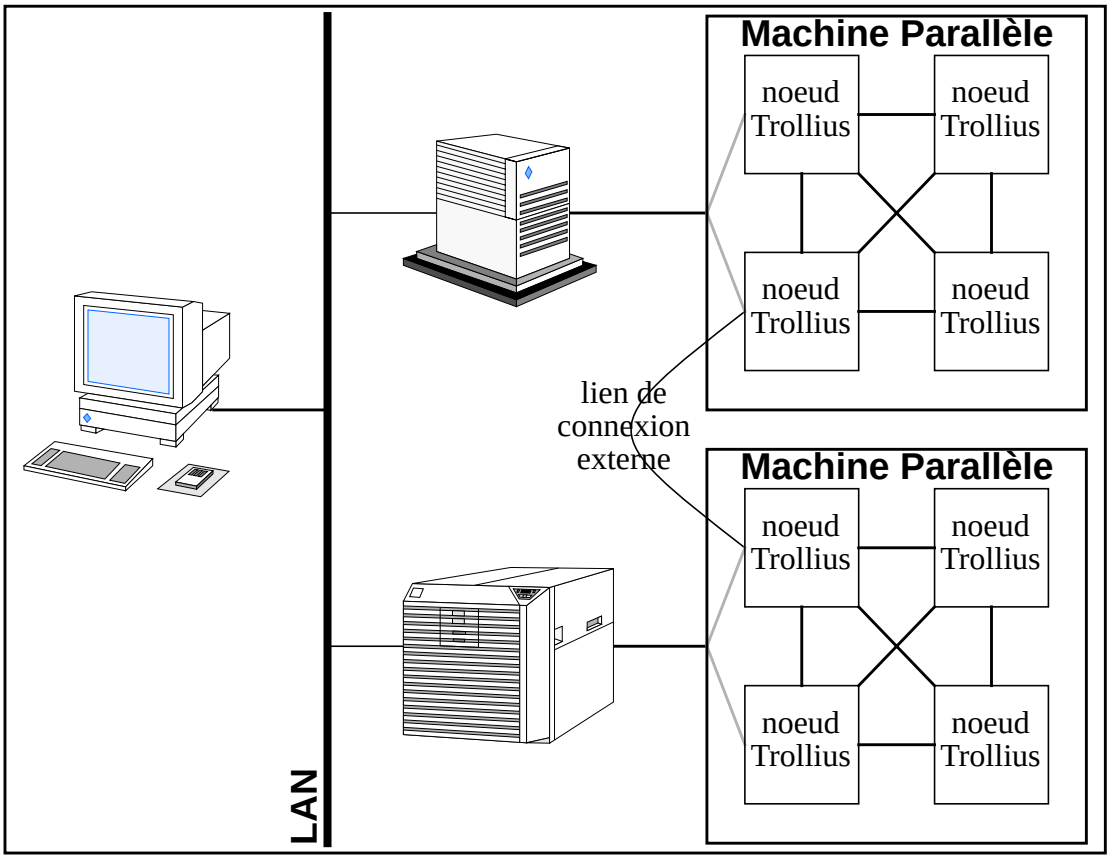


Figure B.7 : architecture matérielle de Trollius

Dans un tel réseau les frontaux et autres ordinateurs monoprocesseurs doivent fonctionner avec Unix; Trollius n'est pour eux qu'une simple interface supplémentaire. Par contre sur les machines parallèles il s'agit d'un noyau système pour chacun des noeuds.

Dans cette architecture Trollius définit un sous-système de communication entre tous les noeuds de toutes les machines, ainsi que d'autres services comme l'accès distant aux fichiers, l'allocation des processus aux processeurs, statique et dynamique. La prise en compte de la reconfiguration des machines parallèles est également une caractéristique importante de ce système. Toutes ces fonctionnalités sont fédérées dans un noyau dont la structure en couches (cf. figure B.8) se distingue nettement des micro-noyaux des systèmes distribués.

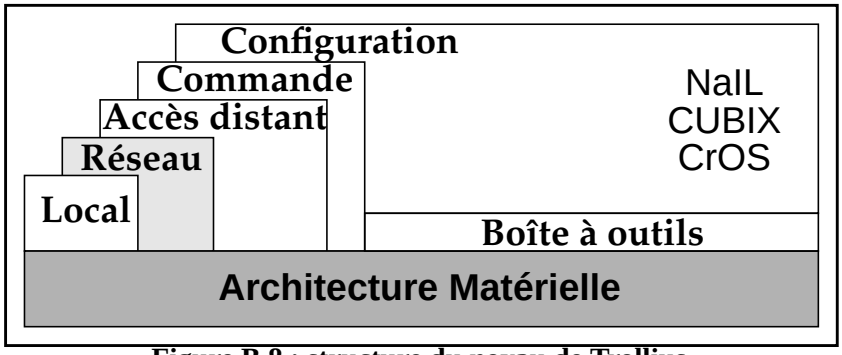


Figure B.8 : structure du noyau de Trollius.

La couche de *Configuration* s'articule autour de l'outil de configuration pour machines parallèles: **NaIL**. A partir d'une description de la topologie d'interconnexion d'une machine et d'un placement initial des processus sur les processeurs, placement fournit par l'utilisateur, NaIL configure le cas échéant le réseau de cette machine, charge le noyau de Trollius sur les noeuds, puis effectue le placement physique des processus (cf. figure B.9).

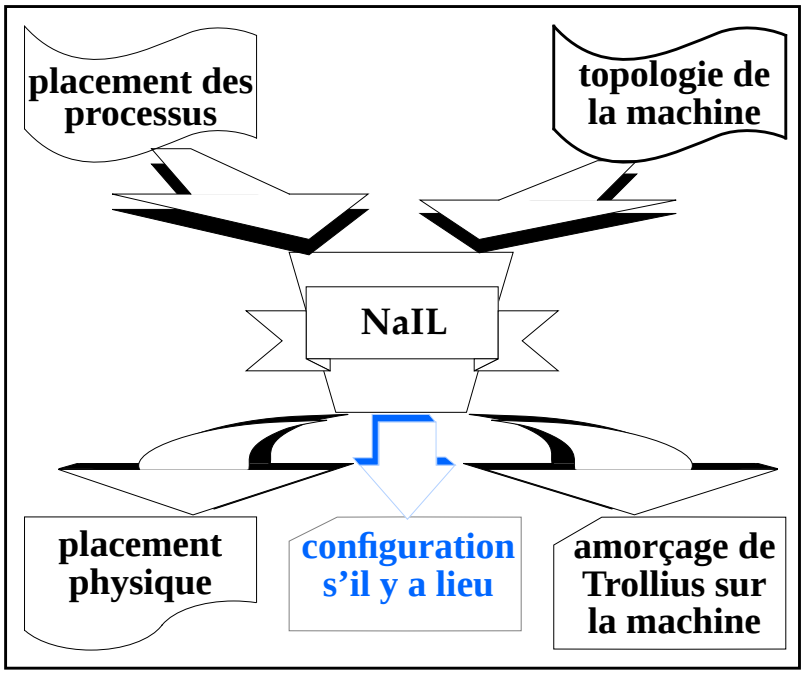


Figure B.9 : fonctionnement de NaIL, l'outil de configuration.

Il est à noter qu'une description de la topologie doit préciser les liens utilisés entre toute paire de processeurs voisins, ainsi que les chemins entre noeuds distants; à une topologie doit donc être associé la fonction de routage qui lui correspond.

A ce niveau des outils de configuration adaptés à des réseaux particuliers aident l'utilisateur dans cette tâche difficile: à titre d'exemple **CrOS** automatise la configuration de grilles. Il existe également un utilitaire, appelé **Crunch**, qui permet d'adapter une machine virtuelle, manipulée par l'utilisateur dans son application, à la machine physique réellement disponible. Crunch agit alors en amont de NaIL.

Le rôle de la couche *Commande* se limite au chargement des programmes sur la (les) machine(s) configurée(s) et amorçée(s), et au contrôle des applications, du noyau et des machines. Lorsque les calculateurs employés ne disposent pas d'une mémoire virtuelle, cette couche fait alors appel à un relocateur d'adresses - ceci résout en partie le problème de la mémoire pour les transputers. L'accès à ces services se fait par l'intermédiaire d'un interprète de commandes.

Trois processus démons assurent les services de la couche *Accès distant*, et sont disponibles dans une bibliothèque de fonctions liée à tout programme. Tout d'abord chaque site hôte exécute le démon **loadd** qui gère le chargement des données et des programmes sur les processeurs. Le contrôle des processus est réalisé par les démons **kenyad** présents sur tous les noeuds, y compris les frontaux. Ces démons permettent non seulement d'exécuter concurremment les processus, mais également de créer, détruire ou accéder à l'état d'autres processus quelles que soient leurs localisations. Notons que Trollius ne considère pas d'autre abstraction que le processus. Enfin, en exécution sur tout noeud qui possède une mémoire secondaire (un disque), le démon **filed** offre aux applications un système de fichiers de type Unix.

La couche *Local* est responsable du contrôle de l'exécution des processus; c'est à ce niveau qu'est réalisé le contrôle des processus. Les protocoles de communication ont donc ici une grande importance. Pour cette couche un message n'est qu'un événement sur lequel doivent se synchroniser des processus, selon un principe propre au protocole. Il s'agit en fait de la partie du noyau chargée de gérer l'ordonnancement des processus et les signaux. Si l'ordonnanceur qu'elle contient peut manipuler des priorités, sa principale caractéristique est qu'il soit possible de l'inhiber pour utiliser un ordonnanceur préexistant: celui d'Unix par exemple, ou encore celui directement intégré dans le matériel - comme c'est le cas des transputers.

La majeure partie du noyau de Trollius est construite autour des fonctionnalités de la couche *Réseau* qui prend en charge la communication. Pour cela chaque noeud dispose d'un serveur d'échange de messages qui assure une synchronisation par rapport à un nombre programmable de messages. Lorsque ce nombre est fixé à 1 message, alors l'émetteur reste bloqué jusqu'à ce que le récepteur ait consommé le message: il y a donc rendez-vous. Avec un nombre de messages supérieur un tampon est alloué et l'émetteur n'est bloqué que si le tampon est plein, et ce jusqu'à ce qu'un emplacement se libère. Comme cette synchronisation ne prend pas en compte les processus appelants, un filtre peut être associé au message pour la rendre plus sélective. De plus le noyau est prévu pour supporter un protocole de diffusion, qui pourrait alors être implémenté efficacement.

Du point de vue de sa réalisation, la couche réseau est construite selon le modèle **OSI**, avec les couches Liaison, Réseau et Transport. Ceci permet d'avoir plusieurs niveaux de communication. Cela est d'autant plus avantageux qu'une application fortement communicante et qui utilise un grand nombre de processeurs peut bénéficier des **circuits virtuels** qu'il est possible d'établir entre deux processus. En effet ce mécanisme de circuits virtuels s'avère souvent efficace du point de vue du temps de communication.

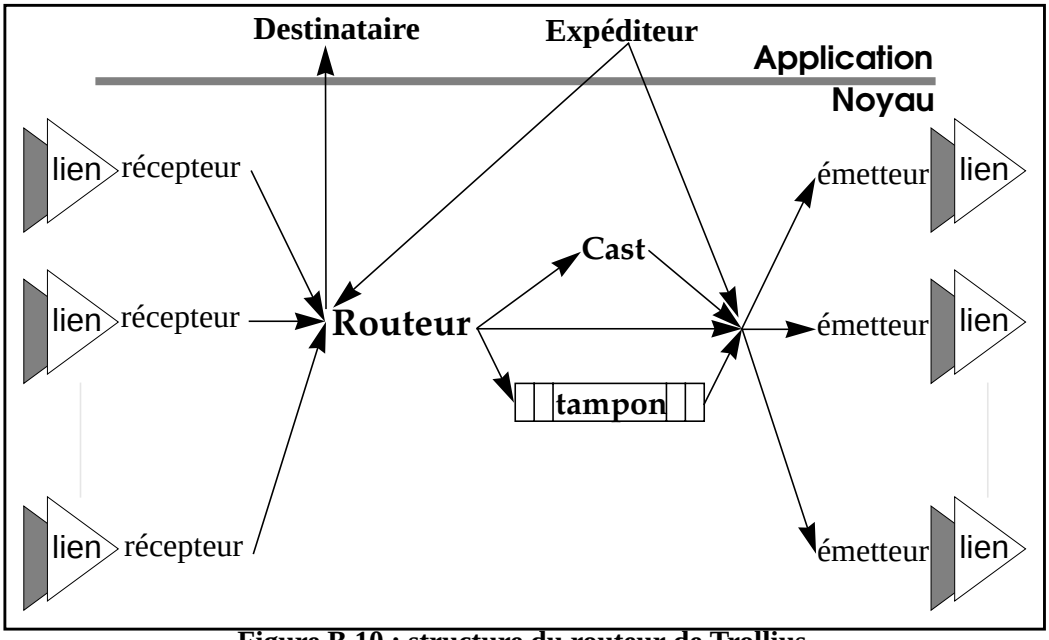


Figure B.10 : structure du routeur de Trollius.

Le routage, nous l'avons vu, doit être en partie contrôlé par l'utilisateur, qui lors de la configuration de la machine par NaIL doit donner les chemins à emprunter entre chaque paire de processeurs. Le routeur de Trollius quant à lui réalise l'acheminement des messages point-à-point et à diffusion (Cast). A cet effet tout lien de communication est gérer par un récepteur et un émetteur de lien. A ces processus viennent s'ajouter le routeur proprement dit qui oriente les messages, un gestionnaire de tampon, et un processus responsable de la diffusion. La figure B.10 ci-dessus représente la structure de ce routeur et les interactions entre ces divers processus.

Un lien de communication peut être de type quelconque: lien transputer, connexion TCP/IP, etc.; ce qui garantit une portabilité aisée et une bonne intégration de diverses machines dans un réseau.

B.4 Peace

Contrairement à Trollius qui s'affranchit des concepts des systèmes distribués, Peace [Berg91, CorSch91, Eich87, SSS89, Schro92] est lui un véritable micro-noyau, dont les concepts sont simplement étendus pour mieux prendre en considération les spécificités variées du parallélisme et notamment celles des architectures massivement parallèles. Ainsi dans son approche le noyau de Peace se base uniquement sur l'échange de messages pour assurer la coopération, le contrôle distribué de tous les objets manipulés par le système, ou la transparence de localisation. Toutefois, il offre une mémoire virtuellement partagée construite au-dessus de ces mécanismes.

Pour obtenir une bonne efficacité du système, dans le contexte des machines massivement parallèles, le noyau de communication doit garantir de hautes performances lors des transferts de données. A cet effet un effort particulier a été apporté sur la réduction des temps de latence lors des échanges de données, facteur important dans toute communication.

Le modèle de processus de Peace se compose de trois niveaux de processus: les **threads**, les **teams** et les **leagues**. Comme allons le voir les deux premiers sont assez similaires de ceux des systèmes répartis, l'innovation est donc d'ajouter un niveau supplémentaire dont nous verrons

l'intérêt. Sans rentrer dans la sémantique associée à chaque niveau, nous pouvons d'ores et déjà dire que cela permet à l'utilisateur d'avoir trois grains de parallélisme: fin, moyen et gros, qui sont fréquemment employés dans la programmation parallèle.

Le grain fin correspond aux processus légers, c'est-à-dire aux ***threads***, qui sont des flots de contrôle séquentiels. Pour le noyau ce sont des unités d'exécution pour lesquelles le contexte d'exécution est très réduit ; ce qui simplifie grandement le changement de contextes et donc accélère leur ordonnancement. Le véritable contexte d'exécution des threads est associé aux ***teams*** (par analogie avec une équipe de sport), qui sont les entités du niveau supérieur. Processus plus lourds, les teams encapsulent un ensemble de threads dans un même espace d'adressage et définissent pour eux un domaine d'exécution. Ce sont également des unités de distribution auxquelles le système associe un placement et une politique d'ordonnancement, politique différente de celle qui est appliquée aux threads.

Jusque là le modèle de processus de Peace ne présente pas de différence notable avec ceux des systèmes répartis ; nous pouvons même dire que cela pourrait suffire pour exploiter le parallélisme des machines, ce qui explique d'ailleurs que les micro-noyaux classiques aient été portés sur certaines machines. Cependant le contexte et l'approche des systèmes parallèles sont biens différents de ceux des systèmes répartis: il y a tout d'abord une différence d'échelle de parallélisme qui est ici beaucoup plus élevée. De plus une application n'est plus considérée comme un processus qui s'attache les services de multiples serveurs, mais plutôt comme une composition de processus qui coopèrent pour accomplir une même tâche et qui s'exécutent en parallèle. En d'autres termes, étant donnée la forte corrélation entre les processus, la notion même d'application manque aux systèmes distribués.

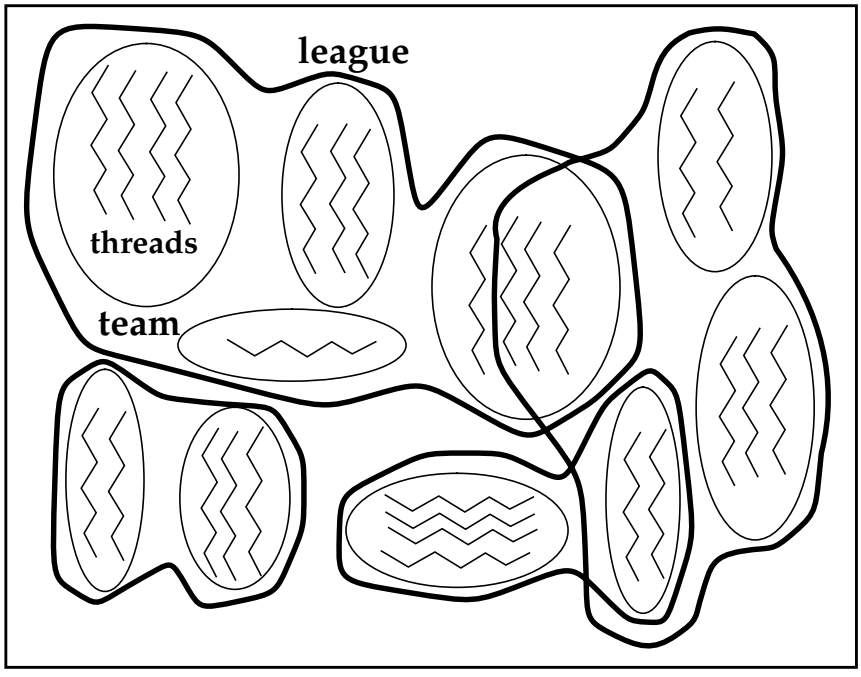


Figure B.11 : modèle de processus de Peace.

C'est ce qu'offre Peace avec les ***leagues*** (par analogie avec une ligue sportive), qui englobent un ensemble de teams dans un domaine de communication. Ce domaine est protégé: aucun échange de messages ne peut franchir les limites d'une league. La league garantit ainsi un fonctionnement sécurisé multi-applications et multi-utilisateurs des machines. Pour permettre une collaboration entre leagues et accéder aux services systèmes qui sont assurés par des lea-

gues spéciales, Peace autorise tout de même le chevauchement de leagues. Ceci ne concerne qu'un ensemble réduit de threads qui servent de points d'accès, les autres threads de la league restant protégés. Le modèle de processus de Peace est illustré par la figure B.11.

Pour communiquer et se synchroniser les teams d'une league, ou plus exactement leurs threads, disposent d'un objet de communication appelé ***gate***[1] qui s'apparente au port des systèmes distribués. Si comme avec les ports tout processus peut y envoyer un message, le fonctionnement d'une gate est toutefois régi par des règles différentes et mieux adaptées au parallélisme. Tout d'abord elles n'ont pas de capacité de stockage dans des tampons, mais sont dotées de possibilités de routage et d'acheminement des messages. Une gate est liée à un unique thread, ou plus précisément à tout thread correspond au moins une gate ; si bien que toute création d'un thread s'accompagne de la création de sa gate. Elles sont également le support de reconstruction dynamique de l'application et de migration des teams. Dans ce cas il est possible d'associer plusieurs gates à un même thread, où elles agissent alors comme des pointeurs de poursuite qui acheminement les messages à l'emplacement effectif du thread.

Même s'il existe plusieurs variantes, un seul protocole est implémenté à la base par les gates: l'appel de procédure à distance. Le modèle de communication de Peace est donc celui du Client-Serveur. Par contre toutes les communications se font en mode synchrone avec rendez-vous entre les entités communicantes. Ceci explique notamment qu'une gate ne soit pas un espace de stockage de messages. Il s'agit donc là d'un mode de communication bien mieux approprié au parallélisme: dans un contexte massivement parallèle l'asynchronisme, en plus du problème de la taille des tampons, engendre des surcoûts, dûs aux recopies de messages inutiles et forts pénalisantes en performances. Cependant ceci ne s'applique que pour des messages de petite taille; le transfert de grands volumes de données est réalisé par un protocole spécifique et mieux adapté qui ne nécessite pas une réelle synchronisation entre les protagonistes de la communication.

Bien que les threads d'une même team peuvent communiquer par l'intermédiaire de leurs gates, ils peuvent également le faire via la mémoire puisqu'ils appartiennent au même espace d'adressage. De plus ils disposent d'événements pour se synchroniser. Un événement est un ensemble de bits, destiné à être placé dans un registre du processeur, qui permet de contrôler l'exécution d'un thread. Dans son fonctionnement le plus simple un thread suspend son exécution jusqu'à ce qu'un autre thread déclenche l'événement désiré. Mais il est également possible d'attendre un certain nombre d'occurrences d'un événement. C'est un mécanisme purement interne à une team, et toute propagation d'événement entre teams est interdite.

Pour accéder et manipuler toutes ces entités Peace met en oeuvre un service de désignation uniforme. A l'intérieur d'une league tout objet est désigné par un identificateur indépendant du type de l'objet. Peace, en fait, ne considère que des objets actifs, et se référer à un objet correspond alors à interagir avec le thread qui lui est associé. Lorsque cet accès se fait au sein d'une même team, seulement un identificateur local est nécessaire: le *lui* (**l**ocal **u**nique **i**dentifier[2]); et puisqu'ils sont fortement corrélés, le *lui* du thread est également celui de sa gate principale. Hors d'une team un *lui*, seul, ne peut avoir de signification; il faut alors lui adjoindre des informations supplémentaires: l'identification du team et une indication du noeud où réside le thread. Ces données forment une *capacité* unique pour toute la league qui donne un accès illimité et transparent à l'objet ainsi désigné. La structure d'un identificateur global est représentée dans la figure B.12.

1. porte.
2. identificateur local unique.

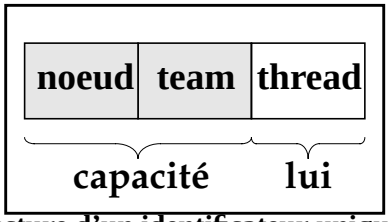


Figure B.12 : structure d'un identificateur unique global de Peace.

La gestion d'une mémoire virtuelle et le partage de mémoire ne sont pas des fonctions de base du système; cependant Peace dispose de ces possibilités, qu'il offre à partir d'un mécanisme d'intégration dans le noyau. Dans ce cas il y a segmentation de la mémoire et pagination des segments; la mémoire se comporte alors comme un cache pour la mémoire secondaire selon le concept des «*fichiers cartographiés en mémoire*»[1] de **Multics** [Orga72]. Les mécanismes qui régissent cette utilisation de la mémoire sont ceux de la copie en écriture (*copy-on-write*) et de la copie sur référence (*copy-on-reference*); ils s'appliquent ici à travers tout le réseau, ce qui permet le transfert d'importants volumes de données.

Pour résoudre un défaut de page le système capte le défaut par déclenchement d'une exception; celle-ci est alors prise en compte par un processus *gardien de la page*[2]. Ce processus, intégré au noyau, tente en premier lieu de résoudre le défaut localement, et en cas d'échec fait appel au *gestionnaire de pages* qui par des échanges de messages a la capacité de retrouver la page dans tout le système. La figure B.13 représente ce fonctionnement. En plus de ces mécanismes Peace offre une mémoire virtuellement partagée, c'est-à-dire le partage de segments entre teams.

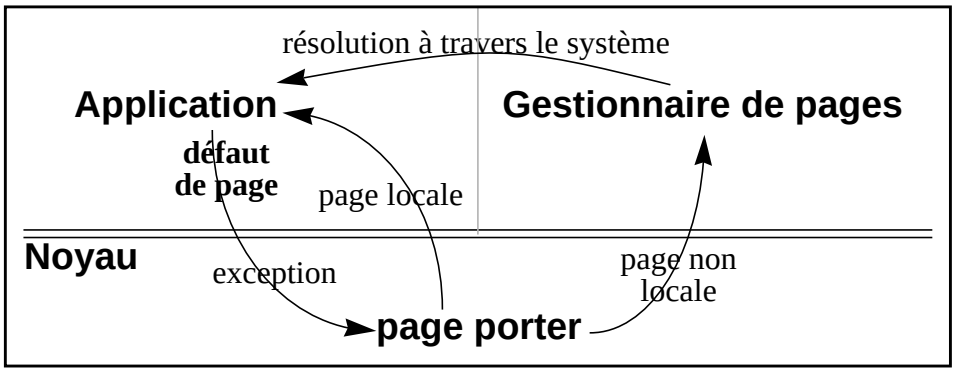


Figure B.13 : traitement d'un défaut de page dans Peace.

Deux cohérences, forte et relâchée, sont disponibles et c'est à l'application de préciser dynamiquement le type de cohérence qu'elle requiert. La cohérence forte nécessite plus d'interactions et a donc une influence non négligeable sur les performances, mais elle garantit que toute lecture fera toujours référence à la valeur la plus récente. Pour obtenir une telle cohérence Peace utilise un protocole d'invalidation qui, lors d'un accès en écriture, révoque tous les droits d'accès à la page partout où elle était dupliquée. Pour éviter des pertes de performances lorsque la fréquence des écritures est élevée, le système dispose d'un mécanisme de cohérence relâchée. Dans ce cas des écritures multiples sur une même page sont autorisées; et c'est alors au compilateur de s'assurer que ces différentes altérations se font sur des parties distinctes de la page. Ce relâchement est momentané, et lorsqu'il y a retour à la cohérence forte chaque site concerné transmet sa modification aux autres qui peuvent ainsi reconstituer la page.

1. memory mapped files.
2. «page porter».

Le système global s'organise autour de trois composants: *POSE*[1] qui offre les services du système d'exploitation parallèle de Peace, et le noyau qui rassemble deux éléments: le *kernel* et le *nucleus*[2]. Comme le montre la figure B.14 ci-dessous, dans l'architecture global du système POSE est construit au-dessus du noyau, et, une application a accès aussi bien aux services de POSE qu'à ceux du noyau.

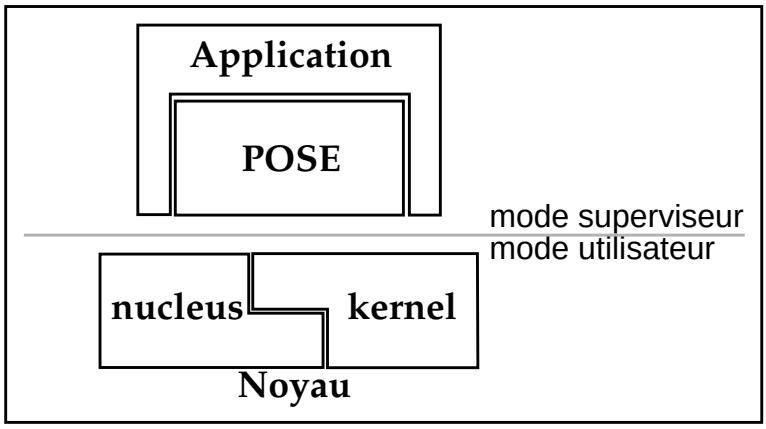


Figure B.14 : architecture du système Peace.

POSE est un ensemble de machines virtuelles qui offre des services systèmes adaptés à une classe d'application, ce peut être la désignation d'objets, le partage de mémoire, ou tout autre service chargé à la demande. Le kernel réalise des fonctions de base: création et destruction de processus, association d'un processus à son espace d'adressage, gestion des interruptions, ou pilotage des périphériques. Chacune de ses fonctions est prise en charge par un thread accessible par RPC, auquel il faut ajouter un thread fantôme[3] supplémentaire qui est le représentant virtuel du noeud pour le service de désignation et le créateur des premiers threads et teams du noeud. Ce kernel est de plus évolutif selon les besoins de l'application, mais aussi et surtout par rapport aux fonctions intégrées au matériel du noeud.

Le nucleus quant à lui gère les abstractions de base de Peace, le modèle de processus et le système de communication. Trois composants forment le système de communication: *NICE*[4], *COSY*[5] et *CLUB*[6]. Le premier définit l'interface du noyau vis-à-vis de l'application et prend en charge l'ordonnancement de bas niveau des processus ainsi que la synchronisation intra-team (les événements). A un niveau inférieur, COSY effectue les communications inter-noeuds et rend NICE totalement indépendant du réseau. Enfin CLUB gère le matériel, pilote les interfaces d'accès au réseau et offre un adressage logique des noeuds. Ces trois éléments fondamentaux du noyaux sont organisés en couches, comme cela est illustré par la figure B.15 ci-dessous.

1. Parallel Operating Services Executive.
2. ces deux termes anglais signifiant noyau, nous les conserverons pour garder une distinction entre l'un et l'autre.
3. de son nom anglais ghost.
4. Network Independant Communication Executive: interface de communication indépendante du réseau.
5. COmmunication SYstem: système de communication.
6. CLuster Bus: pilote des matériels d'accès au réseau - ce nom provient du matériel utilisé dans la machine sur laquelle Peace a été conçu à l'origine: SUPRENUM [Giloi88].

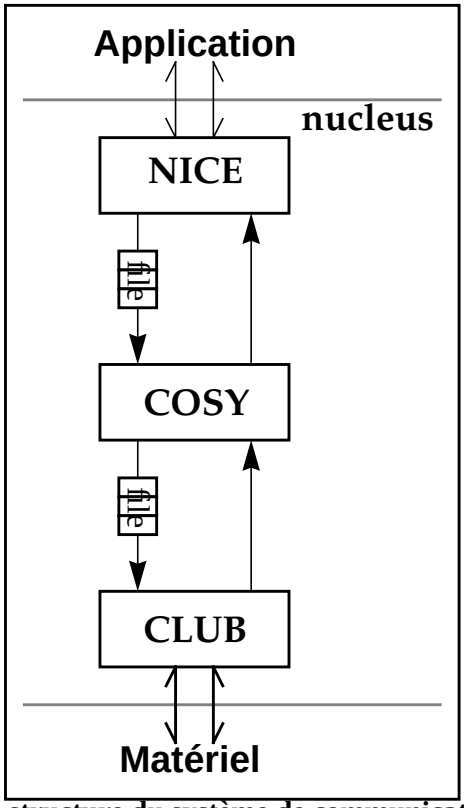


Figure B.15 : structure du système de communication de Peace.

B.5 Comparaison

Une des premières choses que l'on peut retenir de cette description des évolutions des systèmes d'exploitations modernes, est qu'ils ont gagné en souplesse et en rigueur, tant pour leur mise en oeuvre que pour leur utilisation. Leur structure, basée sur un micro-noyau au-dessus duquel peuvent être implémentés divers sous-systèmes, rejoint bien l'un des principaux rôles des systèmes : offrir une abstraction de machine virtuelle pour les différentes classes d'applications.

On remarquera également l'importance de l'échange de messages dans la conception de tous ces micro-noyaux. En effet ce modèle de communication apparait comme le plus fondamental pour rendre les systèmes adaptables aux architectures d'aujourd'hui et à venir, et ce aussi bien sur le plan des architectures réseaux que sur celui des machines parallèles et massivement parallèles. Cette caractéristique rend d'ailleurs les systèmes cohérents avec les langages et les environnements de programmation parallèle, dont nous avons vu précédemment la place primordiale que ces derniers ont fait à l'échange de messages.

Le tableau B.1, ci-après, résume les principales caractéristiques des différents systèmes présentés dans cette annexe, ainsi que celles de PAROS/ParX introduit au chapitre 6. Nous pouvons y distinguer ce qui différencie les systèmes distribués des systèmes parallèles. Si on laisse de côté Trollius, de conception plus ancienne, les systèmes parallèles possèdent en premier lieu un modèle de processus plus évolué, qui avec un niveau supplémentaire encapsule la notion d'application parallèle et celle de domaine de protection. Puis, parce que leurs micro-noyaux ne sont pas figés, les systèmes parallèles ont un modèle de communication évolutif et adaptable. Enfin, seuls les systèmes parallèles font des processeurs des ressources, et de ce fait ils sont mieux adaptés à la répartition de charge - Peace et PAROS disposent d'ailleurs des mécanismes nécessaires à cette fonction. Certes l'approche des systèmes parallèles est essentielle-

ment due aux problèmes posés par les machines parallèles et massivement parallèles, mais il en résulte tout de même qu'elle est plus générale et englobe mieux les architectures existantes.

	Architecture	Modèle de processus	Modèle de communication	Gestion mémoire	Gestion des processeurs
Chorus	μ-noyau	acteurs/ threads	ports asynchrones (client-serveur) groupes de ports	mémoire virtuelle	
Mach	μ-noyau	tâches/ threads	ports asynchrones (client-serveur)	mémoire virtuelle	
Amoeba	μ-noyau	processus/ threads	ports asynchrones (client-serveur) + diffusion	mémoire partagée	
Trollius	en couches modèle OSI	processus	circuits virtuels diffusion		
Peace	μ-noyau évolutif	leagues/ teams/ threads	gates synchrones (client-serveur)	mémoire virtuelle & mémoire partagée	leagues
PAROS	μ-noyau générique	Ptâches/ tâches/ threads	ports et canaux synchrones + machines virtuelles	machine virtuelle	Clusters/ Bunchs

Tableau B.1 : comparatif des systèmes distribués et parallèles.

Annexe C : Algorithmes de PDVM

Pour exprimer le plus clairement possible nos algorithmes, nous avons défini un langage «système» à la Occam adapté au noyau ParX. Tout d'abord nous avons supposé l'existence de types de données structurés: `struct` pour C, ou `record` pour ***Pascal***. Ensuite nous avons introduit les **listes**, dont la déclaration d'une variable est: **`liste de`** `<type> <variable>`, et qui dispose des opérations suivantes: **`Insère`**`(<liste>,<élément>)` pour ajouter un élément à une liste, **`Supprime`**`(<liste>,<élément>)` pour supprimer un élément donné d'une liste, **`Tête`**`(<liste>)` qui renvoie l'élément de tête ou `NIL` si la liste est vide, **`Suivant`**`(<liste>,<élément>)` qui retourne l'élément suivant ou `NIL` s'il n'y en a pas, et, **`Appartient`**`(<liste>,<élément>)` qui retourne l'élément s'il est dans la liste ou `NIL` sinon. Puis Occam a été doté d'opérateurs, d'utilisation similaire à ceux des canaux, sur les ports (objet de communication pour des émetteurs quelconques vers un unique récepteur): ? pour la réception et ! pour l'émission. Enfin, nous avons intégré des opérateurs d'invocation de procédures: `<processeur>!<procédure>(<paramètres>)` pour l'appel d'une procédure dans noeud donné, et `!<procédure>(<paramètres>)` pour l'appel de la même procédure dans tous les noeuds.

C.1 Le protocole de diffusion synchrone

C.1.1 Algorithmes du processus serveur

Fonction d'émission d'un message

```
PROC Srv_sync_bc_send(Requête Req)
  INT grp:
  SEQ
  grp := Req.args.Group
    -- initialisation des paramètres de la diffusion
    Group[grp].Param := Req.args
    Group[grp].NbAck := 0
    Group[grp].ReqEmetteur := Req
    Group[grp].ProcEmetteur := n

    IF
      Group[grp].NbMembres > 0
        Membre courant:
        SEQ
          courant := Tête(Group[grp].Membres)
          WHILE courant <> NIL AND courant <> Req.arg.Ident
            courant := Suivant(Group[grp].Membres,courant)
          IF
            courant <> NIL -- l'émetteur est membre
```

```
                SEQ -- ne pas l'attendre
                  Group[grp].NbPrêts := Group[grp].NbPrêts+1
                  Group[grp].EmetteurMembre := 1
              TRUE
                SKIP
            IF
              Group[grp].NbMembres = Group[grp].NbPrêts
                -- tous les membres locaux sont prêts
                -- donc acquittement local
                Group[grp].NbAck := Group[grp].NbAck+1
              TRUE
                SKIP
          TRUE -- pas de membres locaux, donc acquittement
            Group[grp].NbAck := Group[grp].NbAck+1
        n!Snd_sync_bc_send_first(Req)
:
```

Fonction de réception d'un message

```
PROC Srv_sync_bc_recv(Requête Req)
  VAL INT grp IS Req.args.Group:

  IF
    Appartient(Group[grp].Membres,Req.args.Ident) = NIL
      SEQ -- le processus appelant n'est pas membre
        Req.résultat := SYNC_BC_ERR_NOT_MEMBER
        REVEIL(Req.processus) -- réveil du processus

    TRUE
      SEQ -- un récepteur local de plus
        Group[grp].Nb.Prêts := Group[grp].NbPrêts + 1
        IF
          Group[grp].ProcEmetteur = n
            IF -- la diffusion provient de ce site
              Group[grp].NbPrêts = Group[grp].NbMembres

                SEQ -- tous les membres locaux sont prêts
                  Group[grp].NbAck := Group[grp].Ack + 1
                  IF
                    Group[grp].NbAck = N
                      -- tous les noeuds sont prêts
                      TransmetResteMsg(grp)
                    TRUE
                      SKIP

              TRUE
                SKIP
```

```
            TRUE
              IF
                Group[grp].Param.Taille > 0
                AND Group[grp].NbPrêts = Group[grp].NbMembres
                  SEQ -- il y a une diffusion en attente
                      -- et tous les membres locaux prêts
                    Group[grp].Ack := TRUE -- acquittement
                    n!Snd_sync_bc_ack(Req)
                TRUE
                  SKIP
:
```

C.1.2 Algorithmes du processus émetteur

Fonction d'émission du premier paquet d'un message

```
PROC Snd_sync_bc_send_first(Requête Req)

  [MAX_PACKET_SIZE/4] BYTE paquet:

  SEQ
    paquet := [ Req.args.Msg FROM 0
                FOR min(Req.args.Taille,MAX_PACKET_SIZE/4]
    !Rcv_sync_bc_recv_first( paquet,n,Req.args.Group,
                             Req.args.Taille)
:
```

Fonction d'émission de tous les paquets restants d'un message

```
PROC Snd_sync_bc_send_rest(Requête Req)

  [MAX_PACKET_SIZE] BYTE paquet:
  INT grp, taille, grp, depl, ptaille:

  SEQ

    -- partie déjà envoyée
    depl := min(Req.args.Taille,MAX_PACKET_SIZE/4)

    -- taille restant à envoyer
    taille := Req.args.Taille - depl

    grp := Req.args.Group

    IF
      taille = 0 -- le message a été complètement envoyé
        !Rcv_sync_bc_recv_rest(NIL,0,grp) -- fin diffusion
```

```
        TRUE
          WHILE taille > 0 -- transfert du reste du message
            SEQ
              ptaille := min(taille,MAX_PACKET_SIZE)
              paquet := [Req.args.Msg FROM depl FOR ptaille]
              !Rcv_sync_bc_recv_rest(paquet,ptaille,grp)
              depl := depl + ptaille
              taille := taille + ptaille

      REVEIL(Group[grp].ReqEmetteur.processus)
      ChangePrivilèges(grp)
      RàzSyncGroupParam(grp)
:
```

Fonction d'émission d'un acquittement

```
PROC Snd_sync_bc_send_ack(Requête Req)
  VAl INT grp IS Req.args.Group:
  SEQ
    Group[grp].ProcEmetteur!Rcv_sync_bc_recv(grp)
    Group[grp].Ack := FALSE -- acquittement envoyé
:
```

C.1.3 Algorithmes du processus récepteur

Fonction de réception du premier paquet d'un message

```
PROC Rcv_sync_bc_recv_first(
    [MAX_PACKET_SIZE/4] BYTE paquet,
    INT noeud,grp,taille)

  VAL INT t IS min(taille,MAX_PACKET_SIZE/4):
  VAL Groupe g IS Group[grp]:

  IF
    (g.Param.Taille > 0 AND PlusPrivilégié(grp,noeud))
    OR g.Param.Taille = 0

      SEQ -- diffusion nouvelle ou plus privilégiée
        IF
          g.Param.Taille > 0 AND g.ProcEmetteur = n
            INT prêts: -- diffusion locale à annuler
            SEQ --recalcul des membres prêts
              prêts := g.NbPrêts - g.EmmetteurMembre
              g.résultat := SYNC_BC_ERR_DEADLOCK
              REVEIL(g.ReqEmetteur.processus)
              RàzSyncGroupParam(grp)
              g.NbPrêts := prêts
          TRUE
            SKIP
```

```
          g.TailleReçue := t
          [g.tampon FROM 0 FOR t]:=[paquet FROM 0 FOR t]
          g.Param.Taille := taille
          g.ProcEmetteur := noeud
          IF
            g.NbPrêts = g.NbMembres AND NOT g.Ack
              Requête Req:
              SEQ -- tous les membres locaux sont prêts
                g.Ack := TRUE -- acquittement
                Req.args.Group := grp
                n!Snd_sync_bc_send_ack(Req)
            TRUE
              SKIP

      TRUE -- ignorer cette diffusion
        SKIP
  :
```

Fonction de réception de tous les paquets restants d'un message

```
  PROC Rcv_sync_bc_recv_rest([MAX_PACKET_SIZE] BYTE paquet,
      INT taille,grp)

    VAL Groupe g IS Group[grp]:

    SEQ

      IF
        g.CopieTampon
          SEQ -- recopie du tampon
            CopieAuxMembres(grp,g.tampon,g.TailleReçue,0)
            g.CopieTampon := FALSE

        TRUE
          SKIP

      CopieAuxMembres(grp,paquet,taille,g.TailleReçue)
      g.TailleReçue := g.TailleReçue + taille

      IF
        g.TailleReçue = g.Param.Taille
          SEQ -- message entièrement reçu, fin de la diffusion
            RéveilMembres(grp)
            ChangePrivilèges(grp)
            RàzSyncGroupParam(grp)

        TRUE
          SKIP
  :
```

Fonction de réception d'un acquittement

```
PROC Rcv_sync_bc_recv_ack(INT grp)

  IF
    Group[grp].ProcEmetteur = n -- contrôle de l'ack
      SEQ
        Group[grp].NbAck := Group[grp].NbAck + 1
        IF
          Group[grp].NbAck = N -- tous les noeuds sont prêts
            TransmetResteMsg(grp)
          TRUE
            SKIP

    TRUE
      SKIP
:
```

C.1.4 Primitives utilisées

Nous allons maintenant détailler les primitives les plus importantes parmi celles que nous avons employées plus haut. Il s'agit tout d'abord de la fonction d'activation du transfert d'un message, qui de plus se charge de recopier le message dans les espaces mémoire des récepteurs locaux, c'est-à-dire ceux du site d'émission.

```
PROC TransmetResteMsg(INT grp)

  Membre courant:
  VAL Groupe g IS Group[grp]:

  SEQ

    courant := Tête(g.Membres)

    WHILE courant <> NIL -- copie aux membres locaux
      SEQ
        IF
          courant <> g.Req.args.Ident
              VAL Arguments a IS courant.Req.args:
            SEQ
              a.Msg := [g.Param.Msg FROM a.Depl FOR
                    min(a.Taille,g.Param.Taille-a.Depl)]
              REVEIL(a.courant.Req.processus)
          TRUE -- ne pas encore réveiller l'émetteur
            SKIP

        courant := Suivant(g.Membres,courant)

    n!Snd_sync_bc_send_rest(g.ReqEmetteur)
:
```

La seconde procédure effectue la copie d'un paquet dans les espaces mémoires des récepteurs:

```
PROC CopieAuxMembres(INT grp,[] BYTE src,INT taille,depl)

  INT tailleCopie, débutCopie:
  Membre courant:

  SEQ

    courant := Tête(Group[grp].Membres)

    WHILE courant <> NIL
      VAL Arguments a IS courant.Req.args:
      SEQ
        IF
          a.Taille > 0 AND (depl + taille) > a.Depl
            SEQ
              IF
                depl >= a.Depl
                  SEQ
                    débutCopie,tailleCopie := 0
                    tailleCopie := min(taille,a.Taille)
                TRUE
                  SEQ
                    débutCopie := a.Depl - depl
                    tailleCopie := min( taille-débutCopie,
                                        a.Taille)
              [a.Msg FROM a.Début FOR tailleCopie] :=
                      [src FROM débutCopie FOR tailleCopie]
              a.Taille := a.Taille - tailleCopie
              a.Début := a.Début + tailleCopie

          TRUE -- rien à copier
            SKIP

        courant := Suivant(Group[grp].Membres,courant)
```

Le réveil de tous les membres locaux à un noeud est opéré à partir du même parcours de liste:

```
PROC RéveilMembres(INT grp)

  Membre courant:

  SEQ
    courant := Tête(Group[grp].Membres)
    WHILE courant <> NIL
      SEQ
        REVEIL(courant.Req.processus)
        courant := Suivant(Group[grp].Membres,courant)
:
```

Enfin la procédure et la fonction suivantes manipulent les priorités des sites en cas d'interblocage. Pour obtenir une équité, nous avons utilisé les priorités tournantes.

```
PROC ChangePrivilèges(INT grp)

  VAL Groupe g IS Group[grp]:

  SEQ
    g.PlusPrivilégié := (g.ProcEmetteur + 1) REM N
:

BOOL FUNCTION PlusPrivilégié(INT grp, noeud)

  INT ancien, nouveau:
  BOOL réponse:
  VAL Groupe g IS Group[grp]:

  VALOF
    SEQ
      ancien := g.ProcEmetteur
      IF
        ancien = AUCUN -- pas de diffusion en attente
          réponse := TRUE
        TRUE
          SEQ
            ancien := ((ancien+N) + g.PlusPrivilégié) REM N
            nouveau := ((noeud+N) + g.PlusPrivilégié) REM N
            réponse := (nouveau < ancien)

    RESULT réponse
:
```

C.2 Le protocole de diffusion asynchrone

C.2.1 Algorithme du serveur de requêtes

Dans l'algorithme du processus ci-dessous, ainsi que dans ceux qui suivent nous avons employé la primitive **`sizeof`** qui, comme en C, donne la taille en octets d'une structure de données. De même, nous avons supposé l'existence des primitives **P** et **V** sur les sémaphores.

```
PROC Serveur_Requête(Bcast_Ident id)

  Requête Req:
  INT grp:

  WHILE TRUE
    SEQ

      SyncBcast_receive(  id,ASYN_BC_RQST_GROUP,
                          Req,sizeof(Req))
      grp := Req.Group
      P(Group[grp].TamponSem)

      IF
        Group[grp].Etat = ASYN_BC_PRET
          VAL Groupe g IS Group[grp]:

          SEQ -- pas de diffusion en cours
            g.Etat := ASYN_BC_RECEPTION
            g.Req := Req
            IF
              g.SansTampon
              OR  (g.Tampons <> NIL
              AND Tête(g.Tampons).taille >= Req.Taille)
                -- le groupe n'a pas de tampon ou il y a
                -- assez d'espace, on envoi un acquittement
                ASYN_BC_PORT[Req.ProcEmetteur]!Req.Group
              TRUE -- pas assez d'espace dans le tampon
                g.Etat := ASYN_BC_ATTENTE_TAMPON

        TRUE -- il y a déjà une diffusion en cours
          IF
            Req.ProcEmetteur = n
              Insère(Group[grp].AttenteSnd,Req)
            TRUE
              SKIP

      V(Group[grp].TamponSem)
:
```

C.2.2 Algorithme du serveur d'acquittements

```
PROC Serveur_Ack

  INT grp:

  WHILE TRUE
    SEQ
      ASYN_BC_PORT[n]?grp
      Group[grp].AttenteAck := Group[grp].AttenteAck - 1

      IF
        Group[grp].AttenteAck = 0
          SEQ -- tous les noeuds ont acquittés
            Group[grp].Req.Ack := TRUE -- ack à l'émetteur
            REVEIL(Group[grp].Req.processus)
            Group[grp].AttenteAck := AsynBcastReplication
            SyncBcast_send( NIL,ASYN_BC_CTRL_GROUP,
                            grp,sizeof(grp)) -- activation

        TRUE
          SKIP
:
```

C.2.3 Algorithme du serveur de transfert

```
PROC Serveur_Transfert(Bcast_Ident id)

  INT grp:
  Tampon Bloc:

  WHILE TRUE
    SEQ

      Sync_Bcast_receive( id,ASYN_BC_CTRL_GROUP,
                          grp,sizeof(grp))
      P(Group[grp].TamponSem)
      Group[grp].Etat := ASYN_BC_RECEPTION

      IF
        Group[grp].SansTampon
          VAL Groupe g IS Group[grp]:
          IF
            g.AttenteRcv = NIL -- aucun récepteur prêt
              -- on élimine le message
              SyncBcast_receive(  id,ASYN_BC_XFER_GROUP,
                                  NIL,0)
```

```
        TRUE
          Requête prem, cour:
          INT t:
          SEQ
            prem := Tête(g.AttenteRcv)
            -- réception du message
            SyncBcast_receive(  id,ASYN_BC_XFER_GROUP,
                                prem.Msg,prem.Taille)
            Supprime(g.AttenteRcv,prem)
            -- transmission aux autres destinataires
            WHILE g.AttenteRcv <> NIL
              SEQ
                cour := Tête(g.AttenteRcv)
                t := min(cour.Taille,prem.Taille)
                [cour.Msg FROM 0 FOR t] :=
                                [prem.Msg FROM 0 FOR t]
                REVEIL(cour.processus)
                Supprime(g.AttenteRcv,cour)
            REVEIL(prem.processus)

    TRUE -- groupe avec tampon
      VAL Groupe g IS Group[grp]:
      IF
        g.NbMembres < 1
          -- aucun membre local, élimination
          SyncBcast_receive(  id,ASYN_BC_XFER_GROUP,
                              NIL,0)
        TRUE -- on reçoit le message dans une zone libre
          Requête cour:
          SEQ
            Bloc := Allocation(g.Tampons,g.Req.Taille)
            SyncBcast_receive(  id,ASYN_BC_XFER_GROUP,
                                Bloc.zone,g.Req.Taille)
            g.horloge?Bloc.temps
            Bloc.Req := g.Req
            Bloc.compteur := g.NbMembres
            IF
              g.Req.ProcEmetteur = n
              AND g.Req.Membre <> NIL
                Bloc.compteur := Bloc.compteur - 1
              TRUE
                SKIP
            WHILE g.AttenteRcv <> NIL
              SEQ -- réveil des récepteurs en attente
                cour := Tête(g.AttenteRcv)
                Supprime(g.AttenteRcv,cour)
                REVEIL(cour.processus)
```

```
        IF
          Group[grp].AttenteSnd <> NIL
            Requête cour:
            SEQ -- réveil d'une diffusion locale en attente
              cour := Tête(Group[grp].AttenteSnd)
              REVEIL(cour.processus)
              Supprime(Group[grp].AttenteSnd,cour)
          TRUE
            SKIP

        Group[grp].Etat := ASYN_BC_PRET -- fin du transfert
        V(Group[grp].TamponSem)
    :
```

Bibliographie

[ABG86] *M. Accetta, R. Baron, D. Golub et al.,*
Mach: A New Kernel Foundation for UNIX Development,
Proc. of the Summer 1986 USENIX Conference, pp. 93-112, July 1986.

[Agha86] *Gul Agha,*
Actors: A model of Concurrent Computation in Distributed Systems,
MIT Press, 1986.

[AGP91] *L. Albinson, D. Grabas, P. Piovesan et al.,*
UNIX on a Loosely Coupled Architecture: the CHORUS/MiX Approach,
Proc. of the EIT Workshop on Parallel and Distributed Workstation Systems, Florence, Italy, September 1991.

[AHLR90] *F. Armand, F. Herrmann, J. Lipkis and M. Rozier,*
Multi-threaded Processes in CHORUS/MIX,
Proc. of EEUG Spring'90 Conference, Munich, Germany, pp. 1-13, April 1990.

[AlGot89] *G. S. Almasi and A. Gottlieb,*
Highly Parallel Computing,
The Bejamin/Cummings Publishing Company, Inc., 1989.

[ARG89] *Vadim Abrossimov, Marc Rozier and Michel Gien,*
Generic Virtual Memory Management in Chorus,
Proc. of the «Progress in Distributed Operating Systems and Distributed Systems Management» Workshop, Berlin, Germany, Lecture Notes in Computer Science, Springer Verlag, April 1989.

[ARS89] *Vadim Abrossimov, Marc Rozier and Marc Shapiro,*
Generic Virtual Memory Management for Operating System Kernels,
Proc. of the 12th ACM Symposium on Operating System Principles (SOSP'89), Litchfield Park, Arizona, December 1989.

[BLT91] E. M. Bakker, J. van Leeuwen and R. B. Tan,
Linear Interval Routing,
Algorithm Review, Vol. 2, Nº 2, 1991.

[Bar92] *Geoff Barrett,*
Occam 3 reference manual,
Draft, Inmos, 1992.

[BaMu94] *Alba Balaniuk and Traian Muntean,*
Programming with Shared Data in Parallel Loosely Coupled Machines: the Shared Virtual Memory Approach,
Proc. of the IEEE/USP Int. Workshop on High Performance Computing, Sao Paulo, Brazil, pp129-142, March 1994.

[BBCE94] *V. Bala, J. Bruck, R. Cypher and P. Elustondo,*
CCL: A Portable and Tunable Collective Communication Library,
IBM Technical Report, 1994.

[BCD93] *A. Balaniuk, H. Castro, R. Despons et al.,*
PAROS: A generic multi virtual machines parallel operating system,
Proc. of the Int. Parallel Computing Conference (ParCo'93), Parallel Computing: Trends and Applications, G. R. Joubert et al. (eds.), Elsevier Science, Grenoble, France, September 1993.

[BCD94] *A. Balaniuk, H. Castro, R. Despons et al.,*
Generic construction of virtual machines within the ParX parallel operating system kernel,
Proc. of the IFIP Conf. on Programming Environments for Massively Parallel Distributed Systems, IFIP-WG10.3, Ascona, Suisse, April 1994.

[BCG91] *K. Birman, R. Cooper and B. Gleeson,*
Programming with Process Groups: Group and Multicast Semantics,
Technical Report TR-91-1185, Cornell University, January 1991.

[BDV94] *G. Burns, R. Daoud, J. Vaigl,*
LAM: An Open Cluster Environment for MPI,
Proc. of the Supercomputing Symposium'94, Toronto, Canada, June 1994.

[Benes62] *V. E. Benes,*
On Rearrangeable Three-Stage Connecting Networks,
The Bell System Technical Journal, Vol. XLI, Nº 5, September 1962.

[Berg91] *R. Berg et al.,*
The PEACE Family of Distributed Operating Systems,
Journal of Distributed and Parallel Computing, 1991.

[BFF89] *J. Briat, M. Favre, D. Fort, N. Gonzàlez Velenzuela, Y. Langué, T. Muntean and P. Waille,*
PARX: A Parallel Operating System for Transputer Based Machines,
Proc. of the 10th Occam User Group OUG, IOS Press, Springfield, Amsterdam, April 1989.

[BKS68] *George H. Barnes, Richard M. Brown, Maso Kato, David J. Kuck, Daniel L. Slotnick and Richard A. Stokes,*
The ILLIAC IV Computer,
IEEE Transaction on Computers, Vol. 17, Nº 8, pp. 746-757, August 1968.

[BoSi91] *F. Boussinot and R. de Simone,*
The ESTEREL Language
Another Look at Real Time Programming, Proc. of the IEEE, vol. 79, nº 9, pp. 1293-1304, 1991.

[BRDM90] *G. Burns, V. Radiya, R. Daoud and R. Machiraju,*
All About Trollius,
Occam User's Group Newsletter, 1990.

[Brinch77] *P. Brinch Hansen,*
The architecture of concurrent programs,
Prentice-Hall, 1977.

[BSS91a] *K. Birman, A. Schiper and P. Stephenson,*
Fast Causal Multicast,
Operating Systems Review, pp. 75-79, april 1991.

[BSS91b] *K. Birman, A. Schiper and P. Stephenson,*
Lightweight Causal and Atomic Group Multicast,
Report of the NAG2-593 DARPA/NASA Project, May 1991.

[BuLu92] *R. Butler, E. Lusk,*
User's Guide to the p4 programming system,
ANL-92/17, Argonne National Laboratory, October 1992.

[Burns88] *G. D. Burns et al.,*
Trollius Operating System,
Proc. of the 3rd ACM Conf. on Hypercube Concurrent Computers and Applications, March 1988.

[CaGel89] *N. Carriero and D. Gelernter,*
Linda in Context,
Communications of the ACM, Vol. 32, Nº 4, April 1989.

[CCD94] *C. Castelluccia, I. Chrisment, W. Dabbous, C. Diot, C. Huitema, E. Siegel et R. de Simone,*
Tailored Protocol Development Using ESTEREL,
Rapport de Recherche INRIA Nº 2374, Octobre 1994.

[CCK93] *Peter Carlin, Mani Chandy, Carl Kesselman,*
The Compositional C++ Language Definition,
Technical Report CS-TR-92-02, Revision 0.95, California Institute of Technology, March 1993.

[CEMW93] *H. Castro, A. Elleuch, T. Muntean and P. Waille,*
Generic Microkernel Architecture for the PAROS PARallel Operating System,
Proc. of the World Transputer Congress (WTC'93), Transputer Applications and Systems, R. Grebe et al., IOS Press (Eds.), Vol. 2, September 1993.

[CGRT85] *W. Crowther, J. Goodhue, R. Gurwitz, R. Rettberg and R. Thomas,*
The Butterfly (TM) Parallel Processor,
IEEE Computer Architecture Technical Commitee Newsletter, pp. 18-45, September - December 1985.

[ChaMa84] *J. M. Chang and N. F. Maxemchuk,*
Reliable broadcast protocols,
ACM Transactions on Computer Systems, Vol. 2, Nº 3, August 1984.

[Clos53] *C. Clos,*
A study of non-blocking switching networks,
The Bell System Technical Journal, March 1953.

[CNL89] *S. T. Chanson, G. W. Neufeld, L. Liang,*
A bibliography on multicast and group communications,
ACM Operating Systems Review, Vol. 23, Nº 4, October 1989.

[CorSch91] *J. Cordsen and W. Schröder-Preikschat,*
Object-Oriented Operating Systems Design and the Revival of Program Families,
Proc. of the 2nd Int. Workshop on Object Orientation in Operating Systems (I_WOOOS'91), IEEE 91TH0392-1, pp. 24-28, Pao Alto, CA, October 1991.

[Cray93] *Cray Research Inc.,*
CRAY T3D System Architecture Overview Manual,
Cray Research Inc., 1993.

[DalSei87] *W. J. Dally and C. L. Seitz,*
Deadlock-free message routing in multiprocessor interconnexion networks,
IEEE Transactions on Computers, Vol. c36, Nº 5, 1987.

[DaM78] *Y. K. Dalal and R. M. Metcalfe,*
Reverse Path Forwarding of Broadcast Packets,
Communications of the ACM, Vol. 21, Nº 12, pp. 1040-1048, December 1978.

[DeMu93a] *R. Despons and T. Muntean,*
Constructing Correct Protocols for a Diffusion Virtual Machine in Message Passing Parallel Architectures,
Transputers Applications and Systems'93, Vol. 1, Proc. of the 1993 World Transputer Congress, Aachen, Germany, R. Grebe et al. (Eds.), IOS Press, pp. 465-480, September 1993.

[DeMu93b] *R. Despons et T. Muntean,*
Basic mechanisms for a Parallel Diffusion Machine,
Actes des Journées des Jeunes Chercheurs en Architecture de Machines et de Systèmes, Rennes, France, Décembre 1993.

[**DHN90**] *Mark Debbage, Mark Hill and Denis Nicole,*
Virtual Channel Router Deliverable,
PUMA (ESPRIT P2701) Deliverable Report, October 1990.

[DHN91] *M. Debbage, M. Hill, D. Nicole,*
Virtual Channel Router Version 1.8c User Guide,
Technical Report of the Esprit P2701 PUMA Project, January 1991.

[Dijk68] *E. W. Dijkstra,*
Cooperating sequential processes,
in Programming Langages, F. Gerruys (ed.), Academic Press, pp. 43-112, 1968.

[Dijk75] *E-W. Dijkstra,*
Guarded commands. Non-determinacy and formal verification of programs,
Communications of the ACM, Vol. 18-8, pp. 453-457, August 1975

[DoD82] *Department of Defense,*
Reference Manual for the ADA Programming Language,
revision 1982.

[DTW91] *D. Delesalle, D. Trystram et D. Wenzek,*
Communications on the Connection Machine,
Rapport de Recherche RR848-M, LMC-IMAG, Avril 1991.

[Eich87] *L. Eichler et al.,*
Communication and Management Protocols for the Distributed PEACE Operating System,
Technical Report, GMD-FIRST, Berlin, 1987.

[Elleu94] *Ahmed Elleuch,*
Migration de processus dans les machines massivement parallèles,
Thèse de Doctorat INPG, LGI-IMAG, Novembre 1994.

[FGB91] *A. Forin, D. Golub and B. Bershad,*
An I/O System for Mach,
Proc. of the Usenix Mach Symposium, November 1991.

[FHK91] *G. Fox, S. Hiranadani, K. Kennedy et al.,*
FORTRAN D Language Specification,
Rice University Technical Report TR90-141, April 1991.

[Flynn72] *Michel J. Flynn,*
Some Computer Organizations and their Effectiveness,
IEEE Transactions on Computers, Vol. C-21, 1972.

[ForWyl78] *Steven Fortune and James Wyllie,*
Parallelism in Random Access Machines,
Proc. of the 10th ACM Symposium on Theory of Computing, pp. 114-118, 1978.

[GBD93] *A. Geist, A. Beguelin, J. Dongarra, W. Jiang, R. Manchek, V. Sunderam,*
PVM 3.0 USER'S GUIDE AND REFERENCE MANUAL,
ORNL/TM-12187, Oak Ridge National Laboratory, February 1993.

[GDFR90] *D. Golub, R. Dean, A. Forin and R. Rashid,*
Unix as an Application Program,
Proc. of the USENIX Summer Conference, June 1990.

[Geh84] *N. H. Gehani,*
Broadcasting Sequential Processes (BSP),
IEEE Transactions on Software Engineering, Vol. SE-10, Nº 4, pp. 343-351, July 1984.

[GGT91] *D. Grabas, M. Guillemont, M. Tombroff et al.,*
CHORUS on H1: UNIX System V on Networks of Transputers,
Proc. of Transputing'91, Sunnyvale, CA, April 1991.

[Gian93] *Sylvie Giancone,*
Un modèle de programmation parallèle mixte basé sur l'échange de messages et les données partagées: Réalisation sur une architecture parallèle à mémoire distribuée,
Mémoire d'Ingénieur CNAM., LGI-IMAG, Juin 1993.

[Gien90] *Michel Gien,*
Micro-kernel Architecture - Key to Modern Operating Systems Design,
Unix Review, Vol. 8, Nº 11, November 1990.

[Giloi88] *W. K. Giloi,*
SUPRENUM: A Trensetter in Modern Supercomputer Development,
in Parallel Computing, Vol. 7, Nº 3, pp. 283-296, Proc. of the Int. SUPRENUM Colloquim, Bonn, Germany, Spetember 1988.

[Gonz91] *Néstor Alejandro González Valenzuela,*
PARX: Noyau de système pour ordinateurs massivement parallèles. *Contrôle de la communication entre processus,*
Thèse de Doctorat, INPG, LGI-IMAG, Décembre 1991.

[Guil90] *Marc Guillemont,*
Microkernel Design Yields Real Time in a Distributed Environement,
Computer Technologie Review, pp. 13-19, Winter 1990.

[Hast85] *Robert H. Hastead Jr.,*
Multilisp: A Language for Concurrent Symbolic Computation,
ACM Transactions on Programming Languages and Systems, Vol. 7, nº 4, pp. 501-538, October 1985.

[Hayes86] *J. P. Hayes, T. N. Mudge, Q. F. Stout, S. Colley and J. Palmer,*
Architecture of a Hypercube Supercomputer,
Proc. of the 1986 Int. Conf. on Parallel Processing, pp. 653-660, 1986.

[Heinz93] *Ernst A. Heinz,*
MODULA-3*: AN EFFICIENTLY COMPILABLE EXTENSION OF MODULA-3 FOR PROBLEM-ORIENTED EXPLICITLY PARALLEL PROGRAMMING,
Joint Symposium on Parallel Processing 1993, Waseda University, Tokyo, pp. 269-276, May 1993.

[Hewitt77] *C. Hewitt,*
Viewing control structures as patterns of passing messages,
Journal of Artificial Intelligence, Vol. 8, Nº 3, 1977.

[HJMW86] *J. G. Harp, C. R. Jesshope, T. Muntean and C. Whitby-Strevens,*
The development and application of a low cost high performance multiprocessor machine: Supernode Project,
Proc. of the ESPRIT'86 Int. Conf., Bruxelles, 1986.

[HMA90] *Sabine Habert, Laurence Mosseri and Vadim Abrossimov,*
COOL: Kernel Support for Object-Oriented Environments,
Proc. of OOPSLA'90, Ottawa, Canada, 1990.

[Hoare78] *C. A. R. Hoare,*
Communicating Sequential Processes,
Communications of the ACM, Vol. 21, Nº 8, August 1978.

[HocJes81] *Roger W. Hockney and C. R. Jesshope,*
Parallel Computers,
Adam Hilger Ltd., 1981.

[HOS92] *M. C. Heydemann, J. Opatrny and D. Sotteau,*
Broadcasting and Spanning Trees in de Bruijn and Kautz Networks,
Discrete Applied Mathematics and Combinatorial Operations Research and Computer Science, Vol. 38, 1992

[Hosh86] *Tsutomu Hoshino,*
An Invitation to the World of PAX,
Computer, Vol. 19, pp. 68-78, May 1986.

[HPF93] *High Performance Fortran Forum,*
High Performance Fortran Language Specification,
May, 1993.

[Inmos82] *Inmos Limited,*
The Occam Programming Manual,
Prentice-Hall International Series in Computer Science, 1982.

[Inmos84] *Inmos Limited,*
The Occam2 Reference Manual,
Prentice-Hall International Series in Computer Science, 1984.

[Inmos90a] *Inmos Limited,*
ANSI C toolset reference manual
Inmos D0214-DOCA, August 1990.

[Inmos90b] *Inmos Limited,*
ANSI C toolser user manual,
Inmos D0214-DOCA, August 1990.

[JoBi88] *T. A. Joseph and K. P. Birman,*
Reliable Broadcast Protocols,
Lecture Notes of Artic 88, An Advanced Course on Operating Systems, Tromso, Norway, July 1988.

[JoHo89] *S. L. Johnsson and C. T. Ho,*
Optimum broadcasting and personalized communications in hypercubes,
IEEE Transactions on Computers, Vol. 38, Nº 9, 1989.

[KeSc93] *R. E. Kessler and J. L. Schwarzmeier,*
CRAY T3D: A New Dimension for Cray Research,
Computer Communication Spring'93, San Francisco, California, IEEE Computer Society Press, pp. 176-182, February 1993.

[Kogge81] *Peter M. Kogge,*
The Architecture of Pipelined Computers,
McGrraw-Hill, 1981.

[KTHB89] *M. E. Kaashoek, A. S. Tanenbaum, S. Flynn Hummel and H. E. Bal,*
An efficient reliable broadcast protocol,
Operating Systems Review, Vol. 23, Nº 4, October 1989.

[KTV92] *M. F. Kaashoek, A. S. Tanenbaum and K. Verstoep,*
Using Group Communication to implement a Fault-Tolerant Directory Service,
Internal Report IR-305, Vrije Universiteit, Amsterdam, July 1992.

[Kuskin94] *Jeffrey Kuskin et al.*
The Stanford FALSH Multiprocessor,
Proc. of the 21th IEEE Int. Symp. on Computer Architecture (ISCA'21), Chicago, pp. 302-313, April 1994.

[Lang91] *Yves Bertrand Langué Tsobgny,*
PARX: Architecture du noyau de système d'exploitation parallèle,
Thèse de Doctorat INPG, LGI-IMAG, Décembre 1991.

[Leis92] *Charles E. Leiserson et al.*
The Network Architecture of the Connection Machine CM-5,
Proc. of the 1992 ACM Symposium on Parallel Algorithms and Architectures, 1992.

[Lenoski92] *Daniel Lenoski et al.,*
The Stanford Dash Multiprocessor,
IEEE Computer, pp. 63-79, March 1992.

[Levine82] *Ronald D. Levine,*
Supercomputers,
Scientific American, vol. 246, nº 1, pp. 118-135, January 1982.

[LeTan87] *J. Van Leeuwen and R. B. Tan,*
Interval Routing,
The Computer Journal, Vol. 30, Nº 4, 1987.

[LRMM93] *Y. Lashkari, V. Ramachandran, S. Malpani and S. L. Mehndiratta,*
Vartalaap: a Distributed Multicast Communication System,
Software - Practice and Experience, Vol. 23, Nº 7, pp. 799-811, July 93.

[Menn93] *François Menneteau,*
ParObj: Un Noyau de Système Parallèle à Objets,
Thèse de Doctorat INPG, LGI-IMAG, Mai 1993.

[Merl92] *John Merlin,*
Techniques for the Automatic Parallelisation of 'Distributed Fortran 90',
SNARC 92-02, ESPRIT P2701 (PUMA) Report D4.3.2, University of Southampton, 1992.

[Micha94] *Philippe Michallon,*
SCHEMAS DE COMMUNICATIONS GLOBALES DANS LES RESEAUX DE PROCESSEURS: APPLICATION A LA GRILLE TORIQUE,
Thèse de Doctorat INPG, LMC-IMAG, Février 1994.

[Mign94] *Frédéric Mignard,*
Compilation du langage Esterel en systèmes d'équations booléennes,
Thèse de Doctorat de l'Ecole des Mines de Paris, Octobre 1994.

[Mil89] *R. Milner,*
Communication and concurrency,
Prentice Hall, 1989.

[MMS90] *L. Mugwaneza, T. Muntean and I. Sakho,*
A deadlock-free routing algorithm with network size independent buffering space,
Proc. of CONPAR90-VAPPIV, LNCS 457, H. Burkhart (ed.), pp. 490-501, Zurich, September 1990.

[MPI93] *Message Passing Interface Forum,*
DRAFT Document for a Message-Passing Interface,
November 1993.

[MRT90] *S. J. Mullender, G. van Rossum and A. S. Tanenbaum,*
Amoeba - A Distributed Operating System for the 1990s,
IEEE Computer Magazine, May 1990.

[Mug93] *Léon Mugwaneza,*
Contrôle des Communications dans les Machines Parallèles à Mémoire Distribuée - *Contribution au routage automatique des messages,*
Thèse de Doctorat INPG, LGI-IMAG, Novembre 1993.

[Mull88] *Sape J. Mullender,*
Distributed Operating Systems: State-of-the-Art and Future Directions,
Proc. of the EUTECO 88 Conf., R. Speth (ed.), North-Holland, Vienna, Austria, pp. 57-66, 1988.

[MuWa90] *Traian Muntean et Philippe Waille,*
L'architecture des machines Supernode,
La Lettre du Transputer, Nº 7, pp. 11-40, Septembre 1990.

[MTW93] *M. D. May, P. W. Thompson, P. H. Welch,*
NETWORKS, ROUTERS & TRANSPUTERS: FUNCTION, PERFORMANCE AND APPLICATION,
Inmos, SGS-THOMPSON, IOS Press, 1993.

[Orga72] *E. J. Organick,*
The Multics Operating System,
MIT Press, 1972.

[PouMay88] *Dick Pountain and David May,*
A TUTORIAL INTRODUCTION TO OCCAM PROGRAMMING,
Inmos, BSP Professional Books, March 1988.

[Pra91] *K. V. S. Prasad,*
A Calculus of Broadcasting Systems,
Proc. of TAPSOFT'91, Volume 1: CAAP, Springer Verlag, LNCS 493, April 1991.

[Pra93] *K. V. S. Prasad,*
A Calculus of Value Broadcasts,
Proc. of PARLE'93, Springer Verlag, LNCS 694, 1993.

[Raynal91] *M. Raynal,*
La communication et le temps dans les réseaux et les systèmes répartis,
Ed. Eyrolles, 1991.

[RBF89] *R. Rashid, R. Baron, A. Forin et al.,*
Mach: A Foundation for Open Systems,
Proc. of the 2nd Workshop on Workstation Operating Systems (WWOS2), September 1989.

[RetTho86] *Randall Rettberg and Robert Thomas,*
Contention is no obstacle to shared-memory multiprocessing,
Communications of the ACM, vol 29, nº 12, pp. 1202-1212, December 1986.

[RJO89] *R. Rashid, D. Julin, D. Orr et al.,*
Mach: A System Software Kernel,
Proc. of the 34th Computer Society Int. Conf. COMPCON'89, February 1989.

[Rothnie92] *James Rothnie,*
Overview of the KSR1 Computer System,
Kendall Square Research Report TR 9202001, March 1992.

[RozAbro90] *M. Rozier, V. Abrossimov et al.,*
Overview of the CHORUS Distributed Operating Systems,
Technical Report CS/TR-90-25, Chorus Systèmes, 1990.

[SanKha82] *N. Santoro and R. Khatib,*
Routing without routing tables,
Technical Report SCS-TR-6, School of Computer Science, Carleton University, Ottawa, 1982.

[SaSc89a] *Y. Saad and M. H. Schultz,*
Data communication in parallel architectures,
Parallel Computing, Nº 11, 1989.

[SaSc89b] *Y. Saad and M. H. Schultz,*
Data communication in hypercubes,
Journal of Parallel and Distributed Computing, Nº 6, 1989.

[Schro92] *Wolfgang Schröder-Preikschat,*
Scalable Operating System Design,
in Technical Report 646, «Peace - The Evolution of a Parallel Operating System», GMD-FIRST, Berlin, May 1992.

[ScRi93] *André Schiper and Aleta Ricciardi,*
Virtually-Synchronous Communication Based on a Weak Failure Suspector,
Proc. of the 23th IEEE Int. Symp. on Fault-Tolerant Computing (FTCS-23), Toulouse, June 1993.

[ScSa93] *André Schiper and Alain Sandoz*
Uniform Reliable Multicast in a Virtually Synchronous Environnement,
Proc. of the 13th IEEE Int. Conf. on Distributed Computing Systems (ICDS-13), Pittsburgh, May 1993.

[SezVau94] *André Seznec, Thierry Vauléon,*
ETUDE COMPARATIVE DES ARCHITECTURES DES MICROPROCESSEURS INTEL PENTIUM ET POWERPC 601,
Publication Interne nº 835, IRISA, Juin 1994.

[Shap83] *E. Shapiro,*
A subset of Concurrent Prolog and its Interpreter,
New Generation Computing, vol. 3, 1983.

[Slot82] *D. L. Slotnick,*
The Conception and Development of Parallel Processors - A Personal Memoir,
Annals of the History of Computing, Vol. 4, Nº 1, pp. 20-30, January 1982.

[SML93] *I. Sakho, L. Mugwaneza, and Y. Langué,*
Routing with Compact Routing Tables: ILS for Generalized Meshes,
Research Report RR.93-7, Departement Informatique, Ecole Nationale Supérieure des Mines de Saint-Etienne, Juin 1993.

[SSS89] *M. Sander, H. Schmidt and W. Schröder-Preikschat,*
Naming in the PEACE Distributed Operating System,
Proc. of the Int. Workshop on Communication Networks and Distributed Operating Systems within the Space Environment, pp. 293-302, ESTEC, Noordwijk, The Netherlands, October 1989.

[StoWa90] *Q. F. Stout and B. Wagar,*
Intensive hypercube communication,
Journal of Parallel and Distributed Computing, Nº 10, 1990.

[Talbi93] *El-Ghazali Talbi,*
Allocation de processus sur les architectures parallèles à mémoire distribuée,
Thèse de Doctorat INPG, LGI-IMAG, 1993.

[Tera84] *Teradata Corporation,*
DBC/1012 Data Base Computer Concepts and Facilities,
Document C02-0001-0111, October 1984.

[TKB92] *M. F. Kaashoek, A. S. Tanenbaum and H. E. Bal,*
Parallel Programming using Shared Objects and Broadcasting,
IEEE Computer, Vol. 25, Nº 8, pp. 10-19, August 1992.

[TMC90] *Thinking Machines Corporation,*
Connection Machine Model CM-2 Technical Summary,
1990.

[TMR86] *A. S. Tanenbaum, S. J. Mullender and R. van Renesse,*
Using Sparse Capabilities in a Distributed Operating System,
Proc. of the 6th IEEE Int. Conf. on Distributed Computer Systems, pp. 558-563, 1986.

[Top85] *D. M. Topkis,*
Concurrent broadcast for information dissemination,
IEEE Transactions on Software Engineering, Vol. SE-11, Nº 10, October 1985.

[TouPla90] *A. Touzene et B. Plateau,*
Optimal multinode broadcast on a mesh connected graph with reduced bufferization,
Rapport de Recherche RR825-I, IMAG, Septembre 1990.

[Toura89] *B. Tourancheau*
Algorithmique parallèle pour les machines à mémoire distribuée (applications aux algorithmes matriciels),
Thèse de Doctorat INPG, TIM3-IMAG, Février 1989.

[TucRob88] *Lewis W. Tucker and George G. Robertson,*
Architecture and Applications of the Connection Machine,
Computer, vol. 21, pp. 26-38, August 1988.

[Waille91] *Philippe Waille,*
Architectures Parallèles à Connectique Programmable Reconfigurable et Routage,
Thèse de Doctorat INPG, LGI-IMAG, Septembre 1991.

[Wilcke89] *W. W. Wilcke et al.,*
The IBM Victor Multiprocessor project,
Proc. of the 4th Int. Conf. on Hypercubes, April 1989.

[WulBel72] *William A. Wulf and C. Gordon Bell,*
C.mmp - A multi-mini-processor,
Proc. of the AFIPS 1972 Fall Joint Computer Conference, vol. 41, pp. 765-777, 1972.

Liste des Figures

Liste des Tableaux

Index

H

I

L

M

O

P

R

S

T

V

W