PARX : noyau de système pour les ordinateurs massivement parallèles : contrôle de la communication entre processus

Néstor Alejandro Gonzalez Valenzuela

THESE

présentée par

GONZALEZ VALENZUELA Néstor Alejandro

pour obtenir le titre de **DOCTEUR**

de l'INSTITUT NATIONAL POLYTECHNIQUE de GRENOBLE

(Arrêté ministériel du 23 Novembre 1988)

Spécialité : **Informatique**

PARX : Noyau de système pour les ordinateurs massivement parallèles.

Contrôle de la communication entre processus.

Date de soutenance : 13 Décembre 1991

Composition du jury :

Président	Sacha	KRAKOWIAK
Rapporteurs	Guy	MAZARE
	Michel	RAYNAL
Examinateurs	Roland	BALTER
	Traian	MUNTEAN
	Marc	ROZIER

Thèse préparée au sein du Laboratoire de Génie Informatique

A mi querida esposa Miriam
a mis hijos Miriam Alejandra, Carlos y Roberto
y a mi madre...

... porque los amo a todos simplemente

Remerciements

Je me consacre avec plaisir à cette belle coutume de remercier tous ceux qui nous ont aidé, car je reconnais humblement que cette thèse n'aurait jamais vu le jour sans le concours d'un grand nombre de personnes.

Traian Muntean tout d'abord, à qui j'accorde toute ma reconnaissance pour avoir eu la confiance de m'accepter dans son équipe et pour avoir su me combler d'une amitié sincère. Merci Traian pour m'avoir appuyé sur le plan scientifique aussi bien que sur le plan moral et matériel, sans jamais m'imposer aucune condition, je ne peux que te faire part de mon admiration et mon respect.

Miguel Santana, qui fut le premier à qui j'ai parlé de l'idée de réaliser une thèse à Grenoble. Merci Miguel pour m'avoir hébergé chez toi dès mon arrivée en France et pour m'avoir aidé maintes fois quand j'en avais le plus besoin.

Sacha Krakowiak, pour l'honneur qu'il me fait de présider le jury.

Guy Mazaré et Michel Raynal, pour avoir accepté d'être rapporteurs de ce travail.

Marc Rozier, pour avoir eu la gentillesse de se déplacer depuis Paris pour faire partie du jury.

Roland Balter, qui a démontré une fois de plus sa générosité en acceptant d'examiner ce travail malgré ses contraintes temporelles.

Chacun des membres de l'équipe Systèmes Massivement Parallèles, qui font tous honneur au nom de l'équipe "SYMPA". D'abord les anciens : Yves Langué, Philippe Chol, Philippe Waille, Léon Mugwaneza, Ibrahima Sakho, Ghazali Talbi, François Menneteau , Yu Xiaobo et Pierre Bessiere. Ils ont donné à ce travail un environnement scientifique de très haut niveau. Sans leur précieuse collaboration, toujours amicale et inconditionnelle, j'aurais eu du mal à surmonter les moments difficiles. Puis les nouveaux : Harold Castro, German Vega, Robert Despons, Leila Baccouche et Ahmed Elleuch, qui m'ont beaucoup aidé et surtout encouragé lors de la phase finale de ce travail. Ils sont les héritiers, j'espère qu'ils sauront en profiter.

Catalina Martinazzoli, qui a lu maintes fois le manuscrit à la chasse de mes trop nombreuses fautes d'orthographe et de redaction. Ses remarques, toujours pertinentes, ont beaucoup contribué à l'amélioration du manuscrit. S'il y a encore des imperfections, ce n'est sûrement pas de sa faute.

Table des matières

Liste des Figures

Résumé

Cette thèse aborde un ensemble de problèmes liés à la conception et à la mise en œuvre d'un noyau de communication faisant partie de Parx, un noyau de système d'exploitation pour machines multi-processeurs sans memoire commune, développé dans le cadre du projet de recherche européen ESPRIT "**Supernode**".

Le noyau realise une machine virtuelle, vis-à-vis des communications, dans laquelle l'ensemble de processeurs est complètement connecté indépendamment de la topologie du réseau d'interconnexion sous-jacent. La machine virtuelle offre une interface qui facilite l'exploitation correcte du haut degré de parallélisme physique des machines visées.

Après un état de l'art des architectures d'ordinateurs massivement parallèles, il est proposé un modèle de processus et une structure de noyau de système parallèle. Le modèle est basé sur un ensemble d'entités bien adaptées au contrôle de l'exécution des programmes parallèles composés de processus communicants. Ces entités, qui étendent la notion traditionnelle de processus, intègrent des concepts nouveaux visant la meilleure exploitation de l'architecture physique.

Dans le modèle de processus communicants, ceux-ci ne coopèrent que par échange de messages. Le contrôle, correct et efficace, de la communication et la synchronisation entre processus s'exécutant sur une architecture multi-processeurs sans mémoire commune est le thème central de cette thèse. Notre étude s'oriente vers la conception d'un noyau de communication, pour lequel les problèmes concernent essentiellement le routage des messages sans interblocage dans le réseau de processeurs et les protocoles de communication entre processus adéquats au modèle de programmation utilisé.

Mots clés : parallélisme, système parallèle, architectures parallèles, système réparti, processus communicants, routage des messages, interblocage.

Présentation

Thème central de l'ouvrage

Un intérêt considérable existe actuellement pour l'exploitation du parallélisme ; il se manifeste aussi bien dans la programmation des applications intrinsèquement parallèles que dans la conception d'architectures d'ordinateurs capables de fournir des performances absolues qui dépassent celles des superordinateurs actuels, et qui offrent un meilleur rapport coût-performance.

Parmi le grand nombre d'architectures parallèles proposées récemment, un important effort en recherche et développement est orienté vers une classe d'architectures à haut degré de parallélisme, qui s'obtient grâce à un nombre important de processeurs, chacun ayant une mémoire privée, interconnectés par multiples liens de communication.

Ces ordinateurs, de par leur architecture, sont un support idéal pour l'exécution des applications parallèles. Cependant, leur puissance de calcul contraste fortement avec l'absence évidente d'outils adaptés à leur exploitation. L'absence d'un système, et d'environnements de développement adéquats, est aujourd'hui leur plus grand handicap. Actuellement, les utilisateurs sont très peu assistés, aussi bien lors du développement que lors de l'exécution de leurs programmes parallèles.

L'inexistence de systèmes d'exploitation appropriés aux ordinateurs à haut degré de parallélisme s'explique par la quantité et la complexité des problèmes sous-jacents. Cette thèse aborde quelques-uns de ces problèmes : tout d'abord, la conception d'une structure générale de système d'exploitation pour les architectures multi-processeurs sans mémoire commune, puis la conception et la mise en œuvre des mécanismes de communication entre processus.

L'approche que nous suivrons pour aborder le premier sujet consiste à examiner l'état de l'art, représenté aussi bien par les architectures que par les systèmes

d'exploitation développés actuellement dans le monde, distinguer les sujets de convergence et dégager les défauts les plus importants en ce qui concerne la conception des systèmes. A partir des conclusions de cette étude, nous proposons la structure générale d'un système parallèle adéquat.

Dans une architecture multi-processeurs sans mémoire commune, les entités du système ne coopèrent que par échange de messages ; le contrôle de ces échanges est l'objectif principal des mécanismes de communication entre processus, que nous examinerons en deux étapes : la communication entre processeurs, c'est-à-dire l'acheminement des messages dans le réseau d'interconnexion, et la communication entre processus, c'est-à-dire les protocoles nécessaires aux interactions entre les entités actives du système.

Les principaux apports

Les travaux qui aboutissent à cet ouvrage ne pourraient se réaliser sans l'action unificatrice d'un travail en équipe autour d'un projet global. Ce projet est Parx [LM88, Briat89] et l'équipe est SYMPA[1], dirigée par M. Muntean, au sein du Laboratoire de Génie Informatique de l'Institut d'Informatique et de Mathématiques Appliquées de Grenoble. Chaque fois qu'il sera nécessaire, nous ferons référence aux travaux réalisés par les membres de cette équipe.

La définition de la structure de Parx est un exemple du travail en équipe, les idées sont nées au cours de nombreuses discussions ; Parx est présenté en détail par Yves Langué dans son mémoire de thèse[2]. Notre présentation comporte essentiellement les aspects qui permettront au lecteur de bien comprendre la conception générale de Parx : le modèle de processus et le modèle d'interaction entre processus.

Parx introduit des concepts nouveaux, nécessaires et bien adaptés aussi bien aux besoins de la programmation parallèle qu'à l'exploitation des architectures multi-processeurs sans mémoire commune. L'ensemble des concepts formulés est un tout cohérent vis-à-vis du contrôle correct et efficace de l'exécution des programmes parallèles. Ce que nous présentons sur le contrôle de la communication relève d'un travail personnel, mais pas isolé, dont nous pouvons souligner :

- La conception et la mise en œuvre d'un noyau de communication pour des architectures multi-processeur sans mémoire commune. Ce noyau cor-

[1] SYstèmes Massivement PArallèles.

[2] Yves Langué Tsobgny. *Architecture de noyau de système d'exploitation parallèle.* Thèse Docteur INPG, LGI-IMAG, Grenoble, Décembre 1991

respond à la couche de plus bas niveau de Parx. Les problèmes liés à la communication ont été clairement identifiés, et des solutions sont proposées, notamment en ce qui concerne le routage des messages sans interblocage. Nous proposons un nouvel algorithme de routage sans interblocage, bien adapté aux besoins particuliers des architectures visées.

- Un ensemble de propositions pour contrôler la reconfiguration dynamique du réseau d'interconnexion entre processeurs ; caractéristique de l'architecture Supernode [WM90], que nous utilisons à la fois comme cible et outil d'expérimentation.

  La reconfiguration est aussi bien utilisée pour structurer la machine en vue d'une exploitation plus aisée et plus efficace que pour contrôler la communication entre processeurs. Au cours des différents chapitres, nous examinons les diverses formes de reconfiguration, ses possibles applications et leur relation avec le routage des messages.

- Une structure générique pour l'interprétation des protocoles utilisés dans Parx : elle accepte simultanément différents protocoles dans la même architecture du noyau. Un ensemble de protocoles peut s'insérer modulairement en fonction des besoins ; les interfaces sont bien définies et les contraintes de correction établies.

  L'étude d'un protocole de communication particulier, mais très important pour le modèle de système et les architectures qui nous intéressent : c'est le protocole synchrone, point à point, base du modèle d'interaction entre processus communicants. Nous examinons les aspects conceptuels et pratiques de la réalisation correcte de ce protocole, pour un ensemble quelconque de processus communicants qui s'exécutent sur un réseau quelconque de processeurs.

Plan de l'ouvrage

Cette thèse est divisée en quatre parties. La première, composée de deux chapitres, est consacrée à l'étude des architectures d'ordinateurs parallèles ; on met en évidence les principales approches des architectures massivement parallèles et les besoins de mécanismes de communication associés. Le premier chapitre présente les aspects conceptuels et le second présente quelques ordinateurs, choisis comme représentants des principales approches arhitecturales.

La seconde partie est composée de trois chapitres ; elle est consacrée à l'étude des systèmes d'exploitation parallèles. Nous examinons les concepts impliqués, la réalisation proposée par Parx et une comparaison des systèmes d'exploitation représentatifs de l'état de l'art. Le premier chapitre de cette partie présente les aspects conceptuels, le second est consacré à Parx, et le troisième montre la place de Parx parmi d'autres systèmes d'exploitation développés actuellement dans le monde.

La troisième partie aborde les problèmes de communication. D'abord la virtualisation du matériel, c'est-à-dire l'obtention d'un réseau de communication virtuel, complètement connecté, par acheminement correct des messages entre toute paire de processeurs dans un réseau quelconque. Ensuite, nous étudions les aspects relatifs à la réalisation des modèles d'interaction entre processus par échange de messages : protocoles de communication et synchronisation entre processus, qui sont bâtis en dessus du réseau de communication virtuel. Un premier chapitre présente les aspects concernant l'acheminement des messages, le second présente une étude des problèmes d'interblocage et propose une solution originale, le troisième présente la réalisation des protocoles.

La quatrième partie est consacrée à la présentation de nos travaux de réalisation pratique du noyau de communication dont la conception est présentée in extenso dans la partie trois.

Un résumé, qui précède chaque partie, donne un aperçu plus précis des thèmes traités.

Partie I

Les ordinateurs massivement parallèles

Cette partie est composée de deux chapitres.

Le chapitre 1 est consacré à l'étude des principaux aspects conceptuels concernant les architectures parallèles. Notre objectif n'est pas de décider de la meilleure architecture, mais d'examiner les principales approches architecturales et les tendances actuelles de la recherche. Nous essayons de mettre surtout en évidence les problèmes relatifs à la façon dont les processeurs accèdent aux données et les problèmes de communication qui en découlent.

Le chapitre 2 présente quelques architectures d'ordinateurs parallèles qui représentent l'état de l'art. Nous soulignons la façon dont ils intègrent les concepts examinés dans le chapitre 1. A la fin de ce chapitre nous présentons le Supernode : une architecture multiprocesseurs sans mémoire commune à réseau d'interconnexion dynamiquement reconfigurable, qui est la cible d'expérimentation de nos études.

Chapitre 1

Les concepts architecturaux

1.1 Introduction

Le paragraphe suivant est une traduction libre de ce que disait H. T. Kung à l'occasion du "*Panel on Future Directions in Parallel Computer Architecture*" (voir [Tilb89]) :

> *" Nous faisons du parallélisme tout simplement parce que nous ne pouvons pas surmonter la vitesse de la lumière. Si vous vous contentez d'une machine à un milliard d'instructions par seconde ... alors nous n'avons pas besoin de faire du parallélisme. Mais, si vous souhaitez une machine encore plus performante ... alors nous devons vivre avec le parallélisme. "*

Cette phrase exprime assez bien le sens du parallélisme dans la conception et l'exploitation des ordinateurs parallèles ; effectivement, des puissances de calcul, qui dépassent largement le milliard d'instructions par seconde, sont demandées aujourd'hui par de nombreuses applications scientifiques. La simulation des phénomènes physiques tels que la prévision météorologique, la prédiction du flux océanique, le comportement aérodynamique, entre autres[1] , est normalement limitée en résolution et précision à cause des ressources de traitement informatique insuffisantes. Dans [Bryan89], McBryan assure que ce genre d'application ne pourra vraiment avancer que lorsque des performances de l'ordre de 100 Gflops seront disponibles. La limite se trouve donc dans la capacité de traitement de l'information des superordinateurs actuels et non pas dans la connaissance scientifique ou le modèle mathématique d'un phénomène physique.

[1] La liste n'est certainement pas exhaustive, le lecteur intéressé peut lire par exemple *"Current Developments in Parallel Computation"* de Olivier A. McBryan [Bryan89].

La capacité de traitement de l'information –ou performance– d'un ordinateur, s'exprime aussi bien par la quantité d'opérations en calcul flottant que par la quantité d'instructions qu'il est capable de réaliser par seconde ; ces mesures s'abrègent respectivement en *Mflops*[2] et *Mips*[3].

La *performance de pointe*, qui reflète la capacité de traitement maximale, est utilisée pour comparer et classer les ordinateurs [Hwang87]. La performance réelle, qui correspond à la performance utile mesurée pour une application, se réduit cependant à de valeurs comprises entre 5% et 25% de la performance de pointe. Cette forte réduction est due, d'une part, au fait que des applications différentes s'adaptent mieux à des architectures différentes, et d'autre part, au fait que le temps d'accès mémoire –beaucoup plus important que celui nécessaire aux calculs– limite fortement l'utilisation efficace de l'unité centrale.

Un ordinateur d'aujourd'hui est classé *superordinateur* lorsqu'il atteint une puissance de pointe au-delà de 500 Mflops[4]. Cet ordre de puissance n'était fourni –jusqu'à l'année 85 environ– que par des ordinateurs basés sur un seul processeur dont les représentants les plus puissants sont : NEC SX-2 (1.3 Gflops) ; Hitachi S-810 (840 Mflops) et Fujitsu VP-200 (533Mflops), ou par ceux basés sur une quantité réduite des processeurs très puissants : le Cray 2 (4 processeurs, 2 Gflops).

La performance de ce type d'ordinateur est cependant bornée par certaines limitations physiques : grande dissipation d'énergie dans un volume réduit, vitesse de la lumière et goulot d'étranglement au niveau des accès à la mémoire. Pour faire face à ces inconvénients, les constructeurs ont été obligés d'utiliser des technologies très sophistiquées[5] (certains auteurs les qualifient même d'exotiques [TuRo88]), ce qui augmente considérablement leur coût.

Une solution alternative, radicalement différente, consiste à utiliser des architectures massivement parallèles, dont l'idée de base est que de nombreux processeurs, d'une puissance et d'un coût relativement faibles, coopèrent dans la résolution d'un problème réclamant une grande puissance de calcul.

L'objectif de ce chapitre est de passer en revue les tendances actuelles de la

[2]Un Million de "FLoating point OPerations per Second". Le terme *Gflops* est aussi utilisé pour indiquer un milliard de *flops*.

[3]Un Million d'Instructions Per Second.

[4]Nous suivons ici la classification de [Hwang87].

[5]Par exemple, le Cray 3, qui comportera 16 processeurs pour atteindre une performance de pointe de 16 Gflops, utilise la technologie GaAs.

recherche sur les architectures massivement parallèles auxquelles s'applique notre travail. Nous proposons d'abord un paradigme de parallélisme massif et ensuite nous examinons les aspects qui caractérisent les ordinateurs qui s'adaptent à ce paradigme. L'étude est orientée vers les diverses approches architecturales et met particulièrement en évidence les aspects relatifs à la communication de données entre processeurs.

1.2 Approches architecturales pour le parallélisme massif

Le paradigme du parallélisme massif est le suivant :

> *"Une architecture d'ordinateur est dite massivement parallèle si elle peut aisément s'étendre et fournir la quantité nécessaire de processeurs, donc la puissance de calcul nécessaire, qui permet de résoudre un problème de façon optimale."*

Par ce paradigme, nous voulons souligner que l'intérêt des architectures massivement parallèles ne se trouve pas seulement dans la performance maximale ni dans la quantité absolue de processeurs, mais aussi, et principalement, dans la possibilité de s'adapter aux performances dont une application a besoin et au plus faible coût possible. Cela implique une modularité et flexibilité que les architectures massivement parallèles peuvent offrir de manière presque naturelle. Ce n'est pas le cas des superordinateurs dits classiques, pour lesquels augmenter la performance est à la fois difficile et coûteux.

Il existe aujourd'hui un grand nombre d'ordinateurs parallèles et une diversité d'architectures, la plupart encore dans un état expérimental ou de prototype. Les architectures se distinguent notamment par le nombre et la complexité des processeurs qu'elles utilisent ; le spectre va d'un nombre réduit de processeurs très puissants, comme dans le Cray 2, à des milliers de processeurs très simples, comme dans la Connection Machine [TuRo88]. Les ordinateurs massivement parallèles vont d'une extrême à l'autre ; leur architecture est essentiellement formée par de nombreux processeurs, qui sont parfois de véritables ordinateurs, lesquels communiquent à travers un réseau d'interconnexion.

La grande diversité d'architectures parallèles a donné lieu à de nombreuses propositions de classification[6]; parmi elles, la méthode devenue la plus populaire et la plus utilisée dans la littérature, à tel point que l'on pourrait difficilement parler d'un ordinateur parallèle et de son architecture sans y faire mention, est celle

[6]Une brève description des taxonomies d'architectures parallèles est donnée à la page 32.

proposée par Flynn [Fly72]. Nous utiliserons cette classification pour passer en revue, succinctement, les principales approches architecturales.

1.2.1 Architectures orientées parallélisme des données

Le principe général est que de nombreux processeurs exécutent simultanément la même instruction ; l'ensemble de processeurs est fortement synchronisé, de cette manière, tous ceux qui sont actifs exécutent la même action au même instant. Un mécanisme de contrôle global de la machine, le séquenceur, est alors nécessaire pour gérer le synchronisme.

Les processeurs sont normalement si simples qu'il est possible d'en intégrer un grand nombre dans une seule puce. Chaque processeur a une mémoire privée et couramment un processeur n'accède pas directement à la mémoire des autres. La taille de la mémoire associée à chaque processeur peut être relativement faible car le code exécutable n'est pas stocké par les processeurs, c'est le séquenceur qui diffuse les instructions aux processeurs concernés.

L'architecture est organisée autour d'un réseau d'interconnexion qui permet la communication entre processeurs par l'intermédiaire de liaisons point à point. Aussi bien la structure du réseau, typiquement régulière, que les mécanismes de transfert des données, qui se réduisent normalement à l'échange entre processeurs directement connectés, sont essentiellement déterminés par les besoins qui découlent du type de parallélisme exploité par cette architecture. C'est le parallélisme des données[7], dans lequel l'ensemble des données à traiter est découpé en données élémentaires, chacune traitée par un processeur. Cette décomposition met généralement en évidence des propriétés de localité des données par rapport aux traitements qu'elles sont susceptibles de subir, par conséquent, un processeur donné n'a besoin d'interagir qu'avec un nombre réduit d'autres processeurs qui sont, le plus souvent, ses voisins dans le réseau.

Ces architectures sont couramment appelées SIMD[8]. L'approche SIMD a été utilisée au début pour construire des ordinateurs consacrés à des applications très spécifiques ; le Loral MPP [Batch80], par exemple, fut construit par Goodyear pour le traitement d'images. Cependant, les applications s'exécutant sur ce type d'ordinateur se multiplient [TuRo88], ils ne sont plus vraiment dediés à des applications particulières et donc ils ne sont pas tombés en désuétude comme il est affirmé dans [Tour89]. Certains chercheurs (voir par exemple [GF11])

[7] De l'anglais "*data parallelism*".

[8] *Single Instruction stream- Multiple Data stream.*

n'hésitent même pas à considérer que les architectures SIMD présentent beaucoup d'avantages sur leurs "concurrentes" les architectures MIMD.

1.2.2 Architectures orientées parallélisme des traitements

Le principe fondamental est que chacun des nombreux processeurs peut réaliser un ensemble différent d'actions sur des ensembles de données différentes. Il n'existe pas de mécanisme de contrôle global et chaque processeur agit à sa propre cadence. Le parallélisme exploité est essentiellement le parallélisme des traitements, c'est-à-dire que différents flots d'instructions ont lieu simultanément. Aucune hypothèse n'est faite concernant la localité des données impliquées ni la vitesse relative des traitements. C'est l'asynchronisme qui caractérise le fonctionnement de telles architectures que l'on appele couramment MIMD[9].

Les processeurs sont plus complexes et plus puissants que ceux utilisés dans les architectures SIMD ; ils doivent être capables de stocker et d'exécuter leur propre ensemble d'instructions ; il est donc plus difficile de répliquer plusieurs processeurs sur une même puce. Comme conséquence, le nombre de processeurs utilisés par les architectures MIMD est faible par rapport à une architecture SIMD ; cependant, ils peuvent contenir plusieurs fonctionnalités intégrées dans la même puce, par exemple, dans les transputers, outre l'unité arithmétique et logique, on y trouve une unité de calcul flottant, de la mémoire interne à accès rapide et des fonctionnalités de communication[10] (voir figure 2.6 en page 51).

La dénomination MIMD reste trop générale pour faire état des spécificités de toutes les architectures actuellement en développement. Un critère déterminant pour préciser les caractéristiques des architectures MIMD est la manière dont les processeurs accèdent à la mémoire. Par ce critère, Treleaven (voir l'encadré, page 32) distingue les architectures à *mémoire commune*[11] et les architectures *sans mémoire commune*[12].

Architectures MIMD à mémoire commune

Dans cette classe d'architecture, toute la mémoire est un seul et grand espace d'adressage global et unique, partagé par tous les processeurs. Physiquement, la mémoire est formée par un certain nombre de modules séparés. Un réseau d'interconnexion permet l'accès de chaque processeur à chaque module mémoire ;

[9] *Multiple Instruction stream- Multiple Data stream.*

[10] La nouvelle version annoncée par Inmos pour l'année 1991, le transputer T9000, aura des fonctions d'acheminement des messages intégrées dans le matériel.

[11] Le terme *mémoire partagée* pourrait aussi être utilisé.

[12] Les termes *à mémoire répartie* ou *à mémoire privée* pourraient aussi être utilisés.

un tel réseau correspond généralement à un graphe biparti ayant d'un côté l'ensemble de processeurs et l'ensemble de modules mémoire de l'autre (voir le schéma de la figure 1.2 à la page 33).

Concernant les communications, il n'y a pas d'échange de données entre processeurs, toute communication passe par l'accès à la mémoire partagée, le problème principal consiste alors à maintenir la cohérence des données partagées en évitant les conflits propres aux accès simultanés. Plusieurs processeurs peuvent demander simultanément l'accès à la même adresse mémoire (ou à un même module), des mécanismes de contrôle d'accès sont mis en œuvre dans les réseaux d'interconnexion par des techniques d'entrelaçage, réarrangement, combinaison ou autres. Malgré cela, l'accès simultané à une mémoire commune est encore un goulot d'étranglement et à présent il est admis que des telles architectures ne peuvent s'étendre aisément.

Architectures MIMD sans mémoire commune

Dans ce genre d'architecture, chaque processeur ne connaît que son propre espace d'adressage ; un processeur ne peut accéder à la mémoire d'un autre processeur que par échange de messages à travers un réseau d'interconnexion, celui-ci est différent de celui utilisé pour les architectures à mémoire commune dans le sens où il doit relier tous les processeurs entre eux ; il ne correspond plus à un graphe biparti. La paire processeur-mémoire (P-M), constituant un nœud du réseau, est un tout fortement couplé qui peut être considéré comme étant un ordinateur complet, pour cette raison les chercheurs anglosaxons parlent d'architecture *"multicomputer"* tandis que l'architecture à mémoire commune est dite *"multiprocessor"*. La traduction française de *multicomputer* n'est pas très aisée, on devrait dire *multi-ordinateur* ou *multi-système*, mais aucun des deux termes ne semble donner le sens correct[13].

Dans le parallélisme des traitements, la décomposition d'un problème en différentes unités de traitement permet de mettre en évidence des propriétés structurelles du problème, mais rien ne peut être alors assuré quant à la localité des données par rapport aux traitements. Si l'on rajoute à cela l'asynchronisme entre traitements, on obtient comme conclusion que les problèmes de communication de données entre processeurs sont beaucoup plus complexes que ceux rencontrés dans les architectures examinées précédemment. Dans le cas présent, on ne peut que considérer la situation extrême dans laquelle tous les processeurs ont besoin de communiquer entre eux en totale asynchronie. Le réseau d'interconnexion et

[13] Ces deux dénominations donnent plutôt l'idée des ensembles d'ordinateurs personnels ou des stations de travail reliés par des réseaux locaux, c'est-à-dire des systèmes répartis.

les mécanismes de transfert des messages entre toute paire de processeurs sont alors deux points cruciaux dans cette classe d'architecture.

Le problème des accès simultanés à une mémoire commune ne se pose plus dans les architectures MIMD sans mémoire commune, et malgré les problèmes inhérents au contrôle correct des échanges de messages entre processeurs, que nous examinons en détail dans cette thèse, la taille de la machine peut être aisément modulée ; elle répond très bien à notre définition d'ordinateur massivement parallèle.

Par la suite, lorsque nous parlerons des architectures MIMD sans mémoire commune (NORMA selon Young [Young87]), nous les appellerons aussi *multi-processeurs sans mémoire commune*. Une telle architecture peut être considérée comme étant un réseau de processeurs, la mémoire privée de chaque processeur est implicitement prise en compte. Lorsque nous aurons besoin de nous référer formellement à cette architecture, nous utiliserons la notation suivante : un processeur sera représenté par : p_k, où $0 \leq k \leq N-1$, et N est le nombre total de processeurs du réseau. Les liaisons qui interconnectent les processeurs seront considérées comme étant point à point et bidirectionnelles. La représentation d'une telle liaison pourra prendre l'une des trois formes suivantes, selon l'information spécifique dont on aura besoin :

- l_i : représente le lien numéro i. Le numéro i a été associé au lien indépendamment de l'identification des deux processeurs qu'il connecte.
- l_i^j : représente le lien numéro i du processeur j, il s'agit donc d'une seule extrémité du lien, celle connectée au processeur j.
- $l_{i,j}$: représente le lien qui relie le processeur i au processeur j, dans le sens p_i vers p_j.

Un réseau de processeurs est représenté par un graphe connexe, $RP = (P, L)$ où les sommets $p_k \in P$ sont les processeurs et les arêtes sont les liens reliant les processeurs. Deux processeurs sont dits *voisins* lorsqu'ils sont directement connectés par un lien.

CLASSIFICATION DES ARCHITECTURES PARALLÈLES

		Number of Data Streams	
		Single	Multiple
Number of Instruction Streams	Single	SISD (von Neumann)	SIMD (vector,array)
	Multiple	MISD (pipeline?)	MIMD (multiple micros)

Taxonomie de Flynn

	Execution Stream number and type			
Instruction Stream number and type	Single, Scalar	Single, Array	Multiple, Scalar	Multiple, Array
Single,Scalar	SISSES	SISSEA		
Single,Array		SIASEA		
Multiple,Scalar			MISMES	MISMEA
Multiple,Array				

Taxonomie de Kuck

La taxonomie la plus utilisée pour réferencer l'architecture d'un ordinateur est celle à quatre catégories, proposée par Flynn ; le modèle séquentiel de von Neumann correspond à la classe SISD (Single Instruction-Single Data). Parmi les quatre classes, seulement SIMD et MIMD s'appliquent aux architectures parallèles. La classe MISD ne s'applique à aucune architecture actuelle. Cette taxonomie a été élargie de plusieurs manières ; Kuck, par exemple, remplace les flots de données par flots d'exécution et divise les flots (d'instructions et d'exécution) selon quantité et type. La taxonomie de Kuck vise plutôt le niveau matériel tandis que celle de Flynn vise l'architecture et le jeu d'instruction de la machine. Selon Flynn, l'ILLIAC-IV est une architecture SIMD avec des flots données scalaires multiples ; selon Kuck, un tableau n'est qu'une seule donnée, l'ILLIAC-IV est donc SISSEA. La taxonomie de Kuck est plus détaillée, mais celle de Flynn est beaucoup plus utilisée.

Les architectures MIMD sont elles aussi classifiées de façons diverses ; Treleaven propose une classification selon les mécanismes utilisés pour le contrôle de l'exécution et les accès aux données ; le tableau ci-dessous s'explique par lui-même. Young [Young87], par contre, propose une classification qui tient compte du mode d'accès mémoire : UMA (Uniform Memory Access), NUMA (Non-Uniform Memory access) et NORMA (No Remote Memory Access).

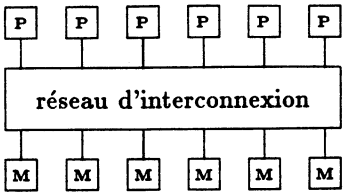
less explicit control ↓

	DATA MECHANISM	
CONTROL MECHANISM	SHARED MEMORY	PRIVATE MEMORY (message passing)
control driven	von Neumann	communicating processes
pattern driven	logic	actors
demand driven	graph reduction	string reduction
data driven	dataflow (I-structure)	dataflow (tokens)

Taxonomie de Treleaven pour architectures MIMD

NOTE: Les trois tableaux ci-dessus ont été repris sans modification du livre "Highly Parallel Computing" des auteurs Almasi et Gottlieb, [AlGot89] pages 111 et 113.

Figure 1.1: Classification des architectures parallèles.



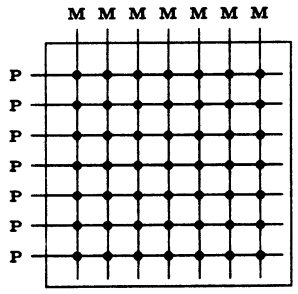
P=processeur ; M=mémoire

Architecture MIMD à mémoire commune :

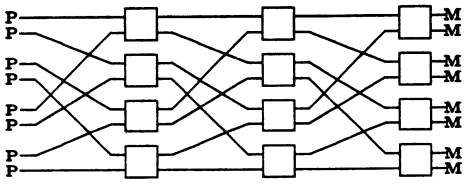
Chaque processeur *voit* tous les modules mémoire à travers le réseau d'interconnexion sans *passer* au travers d'un autre processeur. Les références à la mémoire étant très fréquentes, la performance de cette architecture est très sensible au temps d'accès mémoire ; ce temps doit être minimisé ; il est affecté par le retard introduit par le réseau d'interconnexion et par le fait que plusieurs processeurs peuvent demander l'accès simultané au même module et à la même adresse mémoire : le matériel doit contrôler les accès concurrents en évitant les conflits entre processeurs. Ces deux derniers aspects compliquent sévèrement le réseau d'interconnexion et augmentent son coût, la quantité de processeurs "économiquement utilisables" est limitée à quelques dizaines.

Le réseau d'interconnexion doit permettre à chaque processeur d'accéder à chaque module mémoire, les connexions doivent donc s'établir dynamiquement.

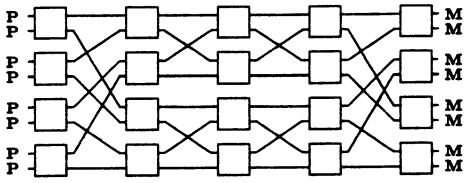
Quelques exemples de réseaux d'interconnexion



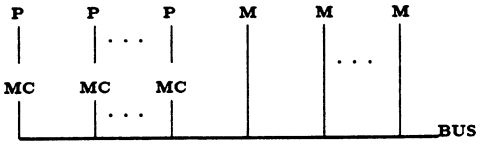
Réseau Crossbar



Réseau Multi-étages Omega



Réseau Multi-étages de Beneš

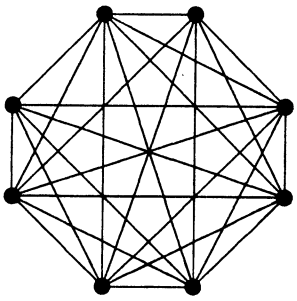


Interconnexion par Bus
(MC=Mémoire Cache)

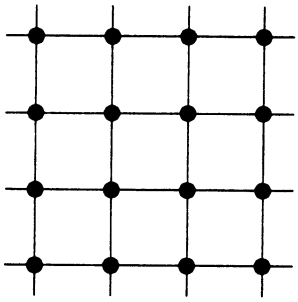
Figure 1.2: Architectures MIMD à mémoire commune.

Quelques exemples de
réseaux d'interconnexion

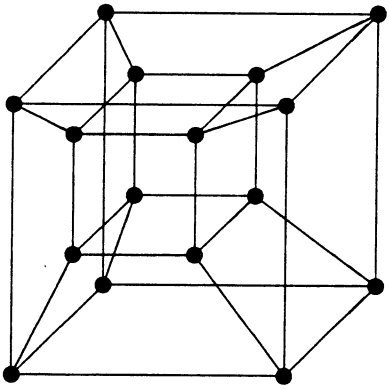
● : répresente la paire P-M



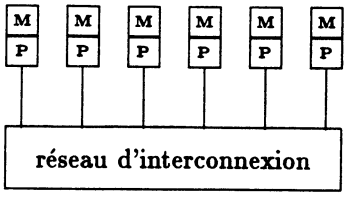
Réseau complet



grille à deux dimensions



hypercube binaire dimension 4



P=processeur ; M=mémoire

Architecture MIMD sans mémoire commune :

Chaque processeur a sa propre mémoire locale (privée), à laquelle il accède directement et efficacement ; contrairement au cas de mémoire commune, un processeur ne peut pas *voir* la mémoire directement à travers le réseau : l'échange de messages entre processeurs à travers le réseau d'interconnexion est le seul moyen dont les processeurs disposent pour accéder à des données non locales. La performance de cette architecture est très sensible au temps de communication entre processeurs ; la structure du réseau d'interconnexion et l'efficacité des moyens de communication sont deux paramètres à prendre en compte. Si des propriétés de localité sont bien exploitées, chaque processeur possède, dans sa mémoire, la plupart des données dont il a besoin, les échanges à travers le réseau sont donc moins fréquents que les accès mémoire dans le cas de mémoire commune. Lorsque le réseau d'interconnexion est basé sur des connexions point à point au lieu d'un bus, des centaines, voir des milliers des processeurs sont ainsi "économiquement utilisables".

Figure 1.3: Architectures MIMD sans mémoire commune

1.3 Réseaux d'interconnexion

Les réseaux d'interconnexion ne sont pas explicitement utilisés dans la caractérisation des architectures parallèles en vue d'une classification. Le réseau d'interconnexion définit cependant la structure spécifique d'un ordinateur ; c'est lui qui caractérise les facilités offertes pour la communication entre les processeurs, et sa participation dans l'activité globale de la machine est si fondamentale, qu'il a une influence cruciale sur ses performances.

Un ensemble de processeurs peut être interconnecté soit par l'intermédiaire d'un *bus*, soit par des liaisons point à point dont chaque processeur n'est directement relié qu'à un certain nombre d'autres processeurs dits voisins. Le bus a l'avantage de servir plusieurs processeurs par une seule voie de communication, il n'existe donc pas la notion de distance entre processeurs. Cependant, la bande passante disponible pour chaque processeur diminue fortement au fur et à mesure que le nombre de processeurs augmente. En pratique, le nombre de processeurs interconnectés par un bus se réduit à quelques dizaines.

Par interconnexion point à point, les processeurs qui sont directement connectés communiquent efficacement car ils peuvent profiter de toute la bande passante offerte par la liaison physique ; néanmoins, l'efficacité diminue lorsque les processeurs communicants ne sont pas directement connectés, en général la bande passante dépend de la distance qui sépare les processeurs, mesurée en nombre de liens à traverser. La notion de distance caractérise le comportement de ce genre de réseau, il est alors essentiel de diminuer son influence. Outre l'augmentation de la vitesse de transmission des liaisons physiques, la recherche des solutions porte sur deux axes : la connectivité des réseaux et les techniques de commutation des données. Le premier est essentiellement une fonction de la topologie du réseau d'interconnexion, le deuxième est lié au routage des messages.

Du point de vue de la distance entre processeurs, le réseau le plus performant est sans doute celui dans lequel tous le processeurs sont directement reliés d'une façon permanente par un lien de communication ; c'est le réseau complètement connecté (voir figure 1.3 à la page 34), mais il est irréalisable pour un nombre de processeurs relativement important ; le nombre de liens par processeur est de $N - 1$, tandis que le nombre total des liens est de l'ordre de N^2.

Pour les architectures massivement parallèles, la minimisation de la distance par une maximisation de la connectivité permanente se heurte à l'impossibilité de réalisation pratique. Des solutions de compromis sont alors nécessaires. Deux approches se dégagent, l'une consiste à connecter les processeurs à travers un

réseau statique[14] , sous une structure régulière qui garantit certaines "bonnes propriétés" telles que : faible diamètre, simplicité du routage des messages et facilité d'expansion. L'autre se base sur des réseaux dits reconfigurables, dans lesquels la distance entre les processeurs communicants peut être modulée par modification dynamique de la connectique du réseau.

1.3.1 Réseaux d'interconnexion statiques

Un grand nombre de structures est proposé dans la littérature ; les plus classiques sont les grilles et les hypercubes. Divers paramètres sont proposés pour caractériser et comparer ces nombreuses structures, la liste que nous proposons ci-dessous donne ceux le plus souvent utilisés.

Lorsqu'il existe plusieurs chemins entre deux processeurs, la *distance* correspond au chemin le plus court.

1. Le *diamètre* du réseau correspond à la distance maximale entre deux processeurs quelconques. Il a une influence sur le temps de communication entre les processeurs non voisins et notamment entre les plus éloignés.

2. La *connectivité* est le nombre de connexions d'un processeur (assimilable au degré d'un nœud dans un graphe). Une haute connectivité rend les communications plus efficaces et facilite l'utilisation de la machine.

3. Le *nombre de liens* nécessaires par processeur ou global au réseau. Il rend compte de la complexité et du coût de réalisation.

4. La *distance moyenne* entre processeurs. Leur intérêt est similaire à celui du diamètre.

5. La *capacité d'expansion* du réseau. Elle rend compte des facilités offertes par la topologie pour étendre le réseau.

D'autres caractéristiques peuvent être aussi considérées, par exemple la tolérance aux pannes (redondance des chemins), la complexité du routage de messages et des fonctionnalités additionnelles (telles que l'entrelaçage de messages). Dans [Wittie81], [WuFeng84], [AgJan86] et [ReFu87] on trouve des études d'évaluation et détermination de paramètres des diverses structures des réseaux.

[14] Le réseau est figé lors de la construction de la machine et n'est jamais changé.

La grille

La topologie d'interconnexion la plus simple est la grille unidimensionnelle, les processeurs sont disposés sur une ligne et chacun est connecté à leurs voisins directs. Un anneau est formé lorsque tous les processeurs ont deux connexions ; lorsqu'une connexion et une seule est éliminée d'un anneau, on a un réseau linéaire dans lequel deux processeurs se trouvent dans les extrêmes avec une seule connexion chacun. Les réseaux à une dimension présentent le plus faible nombre de liens, leur réalisation est très simple, ils s'étendent facilement et le routage des messages suit des règles très simples. Cependant, leur connectivité est réduite. La distance moyenne entre processeurs est de $N/2$ tandis que le diamètre est de l'ordre de N.

On construit des grilles multi-dimensionnelles sur le même principe. La grille la plus utilisée est celle à deux dimensions, car elle présente un bon compromis entre la complexité de réalisation et la connectivité ; les processeurs se trouvant à l'intérieur de la grille ont quatre voisins et ceux se trouvant dans la périphérie en ont trois, exception faite de ceux qui se trouvent dans les quatre coins de la grille qui n'ont que deux voisins. Le nombre maximun de liens par processeur est de quatre pour n'importe quelle taille du réseau, par conséquent, leur coût d'expansion est réduit.

Suivant le même principe de la grille uni-dimensionnelle, si chaque ligne et chaque colonne d'une grille bi-dimensionnelle est en forme d'anneau on a un *tore*, dans lequel tout processeur a quatre voisins. Le diamètre d'une grille de dimension k est de l'ordre de la racine k-ième du nombre de processeurs. De par leur topologie régulière, le routage des messages sur les grilles peut être réalisé de manière simple.

Une grille à deux dimensions est utilisée par l'ILLIAC-IV [Barnes68] (8 processeurs par dimension) et par le Loral MPP (128 processeurs par dimension). Plus récemment, cette topologie à été choisie pour l'architecture de l'ordinateur V256 du projet Victor d'IBM [Wilcke89].

L'hypercube

Les hypercubes se construisent récursivement par duplication ; à partir de deux hypercubes de dimension k, on construit des hypercubes de dimension $k+1$ en reliant les sommets de même emplacement de chaque hypercube. Le nombre de sommets d'un k-cube est 2^k, le nombre d'arêtes est $k2^{k-1}$ et le nombre de voisins de chaque sommet est k. Normalement, on place un processeur sur chaque sommet de l'hypercube, cependant, dans certains cas, comme dans la Connection Machine, le sommet d'un hypercube est une grille ayant 4×4 processeurs.

Cette topologie est très utilisée car elle présente un très bon compromis entre la connectivité, le diamètre et la complexité de réalisation. On la trouve dans plusieurs architectures MIMD sans mémoire commune, par exemple, le *Cosmic Cube* [Seitz85], les iPSC[15], le FPS/T et le NCUBE/10 [Hayes86].

Les structures en hypercube font l'objet de nombreux efforts de recherche : conception de protocoles de communication (notamment la diffusion dans l'hypercube) ; projection d'autres structures topologiques (anneaux, grilles) sur l'hypercube ; algorithmes de résolution de divers types de problèmes numériques, etc., la liste est difficilement épuisable. L'hypercube est donc très populaire, cependant, C. Seitz, le concepteur du *Cosmic Cube*, précurseur des architectures utilisant l'hypercube comme réseau d'interconnexion, préconise actuellement [AtSe88] que le "réseau du futur" est la toute simple grille bi-dimensionnelle. L'explication se trouve d'une part dans les progrès technologiques : la vitesse de transmission des données sur les liaisons physiques est fortement augmentée, et d'autre part, dans les nouvelles idées autour des techniques de commutation des données, comme par exemple "Wormhole" [DaSe86]. Le résultat est que la structure du réseau et les paramètres mesurant la distance entre processeurs, tels que le diamètre, deviennent de moins en moins importants par rapport à la simplicité de réalisation, les possibilités d'expansion et la simplicité de mise en œuvre des fonctions de routage. La même philosophie est à la base du nouveau transputer T9000 (H1) d'Inmos [Pount90].

1.3.2 Réseaux d'interconnexion reconfigurables

Le réseau d'interconnexion idéal permettrait à chaque processeur de disposer d'une connexion physique directe avec tout autre processeur. Nous avons vu que cela n'est pas réalisable par un réseau statique pour un grand nombre de processeurs. D'autre part, les architectures basées sur des réseaux statiques peuvent présenter de bonnes performances pour les algorithmes parallèles qui s'adaptent

[15] Pour Intel Personal Supercomputer.

bien à une architecture particulière, en l'occurrence, celle de la machine physique ciblée par l'algorithme. Leur inconvénient est que, très probablement, beaucoup d'algorithmes ne pourront pas s'adapter convenablement à l'architecture physique et leur résolution sera pénalisée par une perte de performance. Le développement des algorithmes parallèles sous la contrainte de s'adapter à un réseau à topologie fixe est une démarche difficile, et d'ailleurs très peu naturelle : la recherche du meilleur algorithme pour résoudre un problème particulier ne devrait avoir aucune contrainte topologique.

Une architecture dans laquelle le réseau d'interconnexion peut être modifié dynamiquement et configuré en fonction des besoins de communication des applications permet de remédier à ces limitations. La reconfigurabilité permet à la machine, contrairement à l'usage habituel, de se rapprocher des problèmes à résoudre.

Dans les réseaux reconfigurables on ne peut parler ni des topologies ni des paramètres qui caractérisent les réseaux statiques, aucune de ces notions n'existe de par les possibilités de modification de la connectique entre les processeurs, toutes les topologies statiques pourraient éventuellement être réalisées par un réseau reconfigurable.

Parmi les réseaux d'interconnexion qui réalisent un réseau reconfigurable, le plus performant est le *Crossbar* car il permet de connecter, à travers un seul interrupteur logique[16], n'importe quel processeur à n'importe quel autre ; ils permettent notamment de réaliser une correspondance bi-univoque entre leurs entrées et leurs sorties. Si la fonction de correspondance peut être modifiée en fonction des besoins de communication, on peut alors adapter le réseau physique à la topologie d'un problème.

Outre la limitation physique imposée par la quantité des broches admissibles dans un circuit intégré, le problème du réseau Crossbar est que son coût augmente proportionnellement à N^2, son utilisation se limite donc à la connexion d'un nombre relativement réduit de processeurs. Des réseaux dits *multi-étages* sont proposés comme une alternative au réseau Crossbar ; ils sont construits avec des collections des réseaux Crossbar de taille 2×2. Tous les réseaux multi-étages essaient d'approcher le plus près possible les performances du réseau Crossbar ; ils représentent, en général, un compromis dont le coût est réduit de N^2 à $N \log_2 N$, tandis que le nombre d'interrupteurs utilisés pour relier ensemble deux proces-

[16]Un interrupteur est un nœud (ou point) de connexion. Dans le Crossbar, un interrupteur et un seul est utilisé pour relier ensemble deux processeurs. Le délai introduit est alors constant et indépendant de la quantité totale de processeurs.

seurs est augmenté par un facteur de l'ordre de $log_2 N$.

Les réseaux multi-étages sont très utilisés par les architectures multi-processeurs à mémoire commune où chacun des processeurs doit pouvoir accéder à un ensemble de modules mémoire. Le plus utilisé en pratique est le réseau *Omega* ; il se retrouve par exemple dans le NYU Ultracomputer [Gott83], le BBN Buterfly [Crow85] et le RP3 [Pfister86].

Aussi bien le routage des messages dans les réseaux statiques, que la reconfiguration dans les réseaux reconfigurables, ou un mélange approprié des deux[17], émulent le réseau "idéal", c'est-à-dire le réseau complètement connecté, qui présente la connectivité maximale. Pour les programmeurs des applications parallèles, il existe alors un "réseau virtuel" complètement connecté, qui leur permet de programmer sans se soucier de la topologie physique sous-jacente.

1.4 Conclusions

A propos des approches architecturales

L'état actuel du développement des ordinateurs et de la recherche est loin de produire un consensus sur "la meilleure" architecture parallèle. La complexité du sujet et le manque d'une base scientifique solide et universelle, comme le fut –et l'est encore d'ailleurs– le modèle de von Neumann pour les architectures des ordinateurs séquentiels, compliquent fortement l'obtention d'un tel consensus. On constate ceci aisément par les différentes positions, parfois tout simplement contradictoires, que les chercheurs ont sur un même sujet[18] , et d'autre part le grand nombre d'ordinateurs expérimentaux construits sur des modèles de programmation différents. Selon [Bryan88] il aurait autour d'une centaine de projets produisant des prototypes des machines parallèles. Le RP3 d'IBM [Pfister86], par exemple, est un ordinateur conçu exclusivement pour la recherche scientifique portant sur les divers aspects matériels et logiciels du parallélisme ; il vise essentiellement l'expérimentation des divers modes de couplage entre processeurs et mémoires.

A propos des communications

L'échange de messages est le paradigme de base des architectures multi-processeurs sans mémoire commune, par conséquent, la performance de ces architectures est affectée par le temps de transfert des messages entre processeurs dans le réseau.

[17] C'est le cas de l'architecture Supernode présentée dans la section 2.6.

[18] La lecture de l'article [Tilb89] est, dans ce sens, très instructive.

Ce temps est essentiellement affecté par la distance entre les processeurs communicants, des efforts de recherche s'orientent vers l'élimination de l'importance du facteur distance, aussi bien par augmentation de la vitesse des moyens physiques de communication que par la mise en œuvre de nouveaux mécanismes de commutation des données. Cependant, il y aura toujours la nécessité de contrôler l'acheminement correct des messages entre des processurs qui ne sont pas voisins dans un réseau d'interconnexion particulier, c'est le problème du routage des messages, que nous traitons dans le chapitre 6.

Chapitre 2

La mise en œuvre : exemples d'architectures

2.1 Introduction

Ce chapitre a pour objectif de présenter succinctement quelques exemples d'architectures d'ordinateurs parallèles. Le grand nombre de machines existants aujourd'hui rend impossible une présentation exhaustive. Nous avons donc choisi les architectures qui nous semblent être les plus représentatives de l'état de l'art, et qui permettent de souligner les concepts que nous avons examiné dans le chapitre précédent.

Pour chaque architecture, nous donnons un bref aperçu général suivi de la description de la structure d'un nœud de la machine et du réseau d'interconnexion.

2.2 La Connection Machine

La Connection Machine est l'ordinateur SIMD le plus performant aujourd'hui. Disponible depuis 1985, il peut avoir jusqu'à 64K processeurs[1] ; la version CM-2 décrite dans [TuRo88] atteindrait une performance réelle supérieure à 20 Gflops pour une application d'évaluation de polynômes.

Structure d'un nœud

Un processeur de base est constitué d'une unité arithmétique et logique ayant trois bits en entrée et délivrant un résultat en deux bits de sortie (pas plus qu'un

[1] 1K = 1024

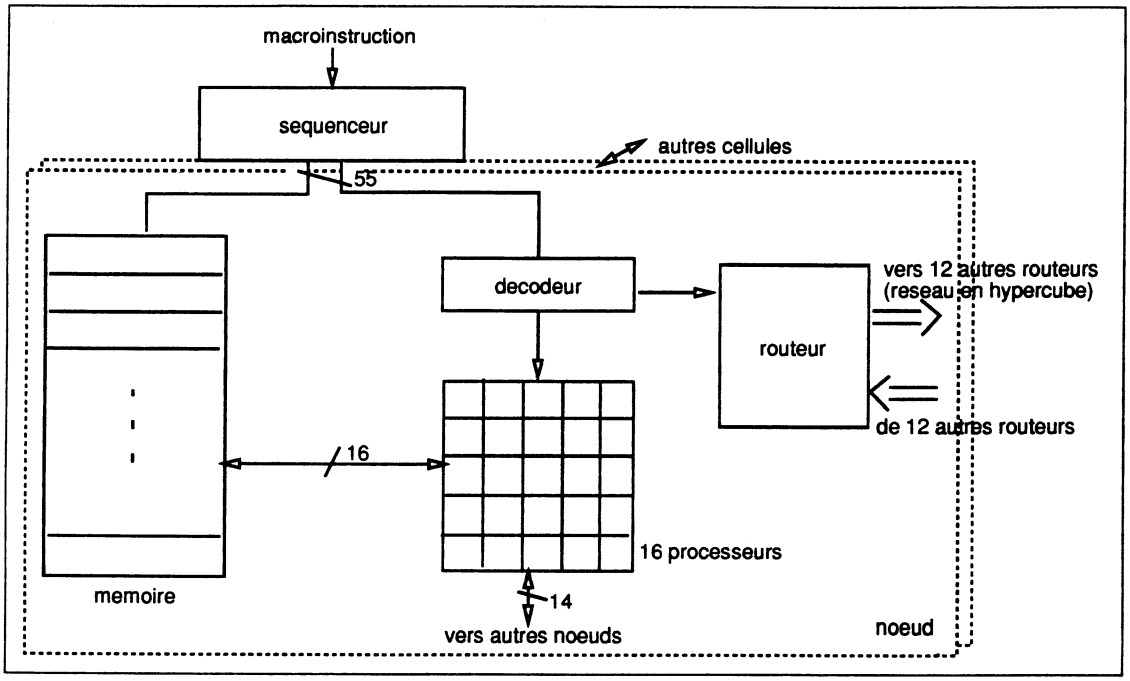


Figure 2.1: Un nœud de la Connection Machine (inspiré de [AlGot89])

simple additionneur logique). Cette simplicité extrême permet d'intégrer 16 processeurs plus un mécanisme de routage câblé dans une seule puce (de 1 cm^2) ayant 68 broches, qui constitue un nœud de la machine. Dans la version CM-1, l'opération de base d'un processeur consiste à lire de la mémoire deux bits A et B et un bit d'un registre F (*"Flag"*), réaliser une opération du type **AND**, **ADD**, **OR**, ou encore **MOVE** et modifier la valeur de A et/ou de F selon l'opération effectuée. Une opération n'induit même pas un calcul logique, elle consiste simplement à "regarder" un registre qui contient la table de vérité de la fonction.

La version CM-2 modifie de très peu le processeur de base, par contre, un processeur de calcul flottant est rajouté tous les deux nœuds, ainsi, 32 processeurs de base partagent un processeur calcul flottant. Chaque processeur accède directement à une mémoire privée de 64 Kbits.

Réseau d'interconnexion

Les 16 processeurs d'un nœud sont interconnectés par un réseau en forme de grille 4×4. A l'intérieur d'un nœud chaque processeur communique directement avec ses 4 voisins immédiats. La communication entre nœuds différents se fait à travers les routeurs connectés en hypercube dont la dimension maximale est 12.

2.3 Le RP3 d'IBM

Le RP3 est un ordinateur expérimental construit par IBM. Bien qu'il puisse être programmé en utilisant l'échange de messages, la mémoire commune reste sa caractéristique prédominante. Il a été conçu pour une configuration maximale de 512 processeurs (en realité 512 véritables ordinateurs) ; fournissant une performance de pointe de 800 Mflops, 1300 Mips et pouvant avoir jusqu'à 4 Gigaoctets de mémoire.

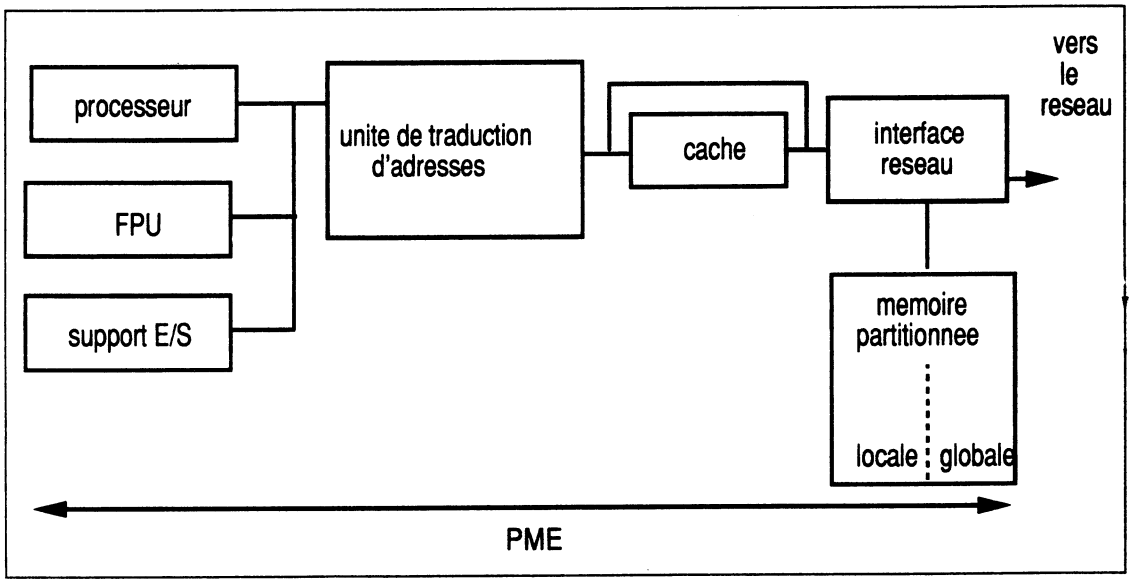


Figure 2.2: Un nœud de la machine RP3

Structure d'un noeud

Chaque nœud est un véritable ordinateur qui contient : un processeur 32 bits ; 32 Koctets de mémoire cache ; un coprocesseur de calcul flottant et 4 Moctets de mémoire, laquelle est dynamiquement partagée en mémoire privée et mémoire commune (la frontière de la partition est modifiable).

Un aspect très intéressant est la façon dont le RP3 traite les accès mémoire : un mécanisme d'adressage transforme une adresse virtuelle en une adresse réelle pouvant se trouver associée à n'importe lequel des 512 nœuds. Les adresses émises par un processeur sont interprétées par l'interface réseau qui peut les aiguiller soit vers la portion de mémoire locale, soit vers le réseau, auquel cas l'adresse est transmise à l'unité d'interface réseau d'un autre nœud où une case de sa portion mémoire commune sera identifiée. Ce mécanisme est bien décrit dans [AlGot89, p464].

Réseau d'interconnexion

Le RP3 comporte deux réseaux différents : l'un dit à "haute vitesse", construit en technologie ECL à 20 nanosecondes/cycle d'horloge. L'autre est dit à "basse vitesse", construit en technologie CMOS ; il a la particularité de combiner des messages faisant référence à la même adresse mémoire. Ainsi, un seul accès permet d'obtenir l'information requise "en même temps" par plusieurs processeurs.

Le premier réseau est composé de quatre étages de commutateurs 4×4, il s'agit d'un réseau multi-étages de Banyan ([WuFeng84] pages 100-107). Le second est un réseau *Omega* dont les points de commutation ont la capacité de combiner des messages. Des mécanismes de commutation de données similaires au *"Wormhole"* (voir la section 6.2.2 à la page 119 de cette thèse) y sont utilisés ; le choix entre les deux réseaux est déterminé par le type d'accès : lorsqu'une requête d'adresse est soumise à l'un des réseaux, elle traverse des commutateurs en *pipeline* vers le nœud destination, lorsqu'elle est bloquée dans un étage (les réseaux Banyan et Omega sont réarrangeables mais bloquants), la requête est conservée dans les différents étages déjà traversés en attendant la libération des connexions en aval de son chemin.

Il faut remarquer que la structure générale des ordinateurs MIMD à mémoire commune et sans mémoire commune, illustrés dans les figures 1.2 et 1.3 ne s'applique pas directement au RP3 ; en effet, celui-ci réalise un mélange des deux : chaque processeur a une mémoire locale "moitié privée, moitié commune", un processeur qui veut accéder à une mémoire non locale à son nœud n'a pas à "envoyer un message" à un autre processeur, les mémoires ne sont pas "cachées derrière" les processeurs, elles sont directement visibles par tous les processeurs à travers le réseau d'interconnexion.

2.4 L'architecture iWarp

La machine iWarp [Bork89], est le fruit de la collaboration entre la compagnie Intel Corporation et l'Université Carnegie Mellon. Un premier prototype est disponible en 1989. iWarp est résolument destiné aux applications hautement parallèles, conciliant de très fortes puissances de traitement et de communication à des possibilités d'entrées-sorties importantes.

Structure d'un nœud

Chaque nœud comporte un processeur de calcul possédant une mémoire sur la puce, de la mémoire locale et un processeur de communication. La puissance de

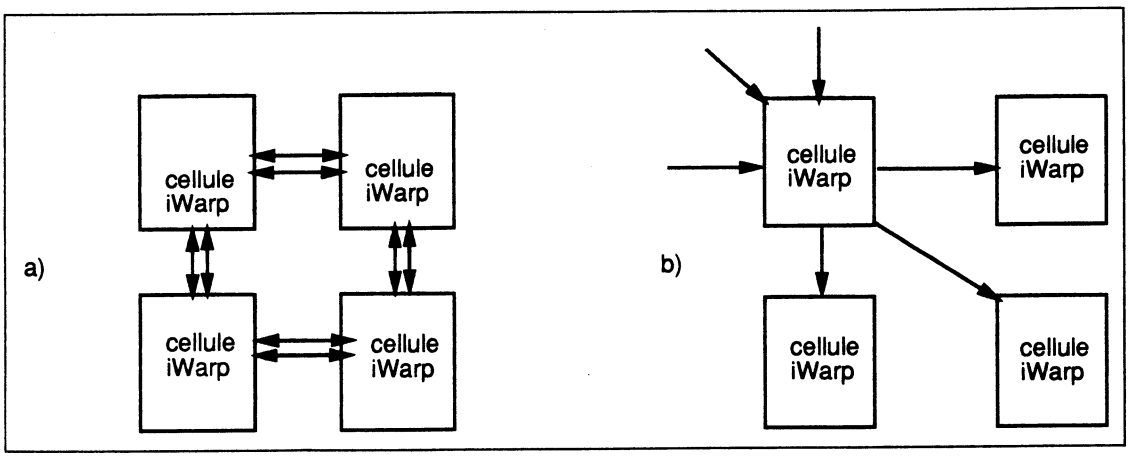


Figure 2.3: Différents réseaux pour la machine iWarp

traitement (20 Mflops, 20 Mips) est associée à une forte puissance de communication (4 ports entrants, 4 sortants, débit total 8 $\times$ 40 Mo/s= 320 Mo/s, temps de latence de 100 à 150 ns). Les deux processeurs fonctionnent en parallèle et communiquent par une file d'attente de registres. Les ports de différents nœuds peuvent être reliés point à point par un bus unidirectionnel.

Le matériel fournit le support pour le routage "Wormhole" selon une stratégie d'acheminement baptisée "streetsign". Le comportement général est l'ouverture d'un chemin pour chaque message. Lors de cette ouverture, le cheminement du message est indiqué à l'aide d'un suite de couples (nom de cellule, action suivante). Les actions s'assimilent aux indications de la signalisation routière : stop, aller à droite, etc.

Les bus physiques sont multiplexés en un certain nombre de bus logiques (jusqu'à 20), chacun exclusivement attribué à un chemin. Chaque mot d'un message est lu et immédiatement réémis sur le bus logique suivant sur son chemin (c'est la technique du "Wormhole"). Le processeur de communication supporte ainsi le routage dans des réseaux de dimension 1 ou 2 (4 ports sortants).

Le réseau d'interconnexion

Conformément à l'objectif de faire exécuter des applications à grain de communication fin (importance accordée aux algorithmes systoliques), et d'autres à grain plus gros (algorithmes de vision), iWarp offre un support pour des réseaux en grilles de dimension 1 et 2, ou des réseaux spécialisés (voir figure 2.3). Dans les premiers, les ports de chaque cellule sont regroupés par couples {entrée, sortie} pour être reliés à une cellule voisine par une paire de bus. Dans les seconds, les

ports sont découplés et l'on peut ainsi réaliser un treillis hexagonal par exemple.

Deux modèles de communication sont proposés : l'échange de messages et la communication dite "systolique". L'échange de messages se caractérise par l'accumulation d'une information, puis son transfert en tant qu'unité. Dans la communication systolique, l'information n'est pas accumulée mais transmise à son destinataire au fur et à mesure qu'elle est produite. iWarp propose des chemins privés correspondant à un multiplexage des bus physiques. Un message réserve un chemin et se termine par un marqueur de fin qui libère le chemin.

2.5 Le V256 d'IBM

La machine V256 du projet Victor d'IBM [Wilcke89] est un exemple typique des architectures MIMD sans mémoire commune basées sur des réseaux d'interconnexion à topologie statique.

La machine est partitionnée entre différents utilisateurs qui peuvent mettre en œuvre des environnements de programmation différents. Aucun paradigme d'utilisation du parallélisme n'est imposé. Deux axes de recherche sont cependant explorés. Le premier est la parallélisation des opérations d'entrées-sorties, essentiellement à l'usage du système de fichiers. Le second est la projection des graphes de tâches usagers sur une configuration de la machine, avec pour objectif d'aboutir à un outil automatique.

Structure d'un nœud

Un nœud de la machine V256 est constitué d'un transputer T800, dont nous parlerons plus en détail dans la section suivante. En accord avec la philosophie des multi-processeurs sans mémoire commune, chaque processeur exécute son propre flot de contrôle d'instructions sur des données privées non partagées. Les instructions et les données sont contenues dans la mémoire locale de chaque processeur.

Le réseau d'interconnexion

La machine V256 est formée d'une grille 16×16, soit un total de 256 processeurs (voir figure 2.4). Dans le choix de la topologie du V256, le faible nombre de liens physiques du processeur utilisé est une restriction. Cependant, ses constructeurs ont préféré une topologie régulière plutôt qu'une topologie irrégulière de plus faible diamètre.

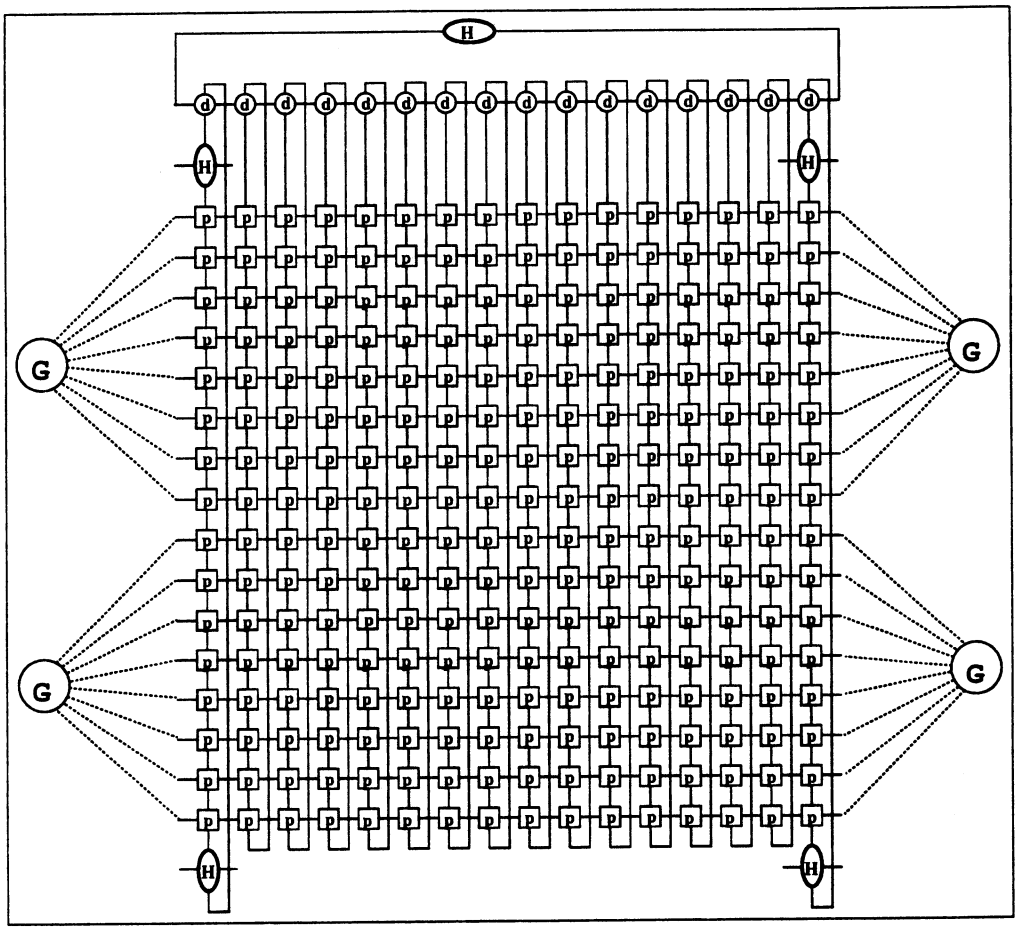


Figure 2.4: Architecture du Victor V256 d'IBM.
G : station graphique, H : machines hotes
d : processeurs supportant des disques, p : processeurs de calcul.

Le V256 est une machine multi-utilisateurs, et les contraintes liées à la sécurité des usagers entraînent un ensemble de dispositifs matériels. La machine est partitionnée entre ses utilisateurs, aucun processeur n'étant partagé. Les différents sous-réseaux ainsi créés n'interfèrent pas les uns dans les autres.

Les ressources de la machine sont gérées pour tous les utilisateurs par des serveurs systèmes. L'architecture distingue donc clairement des processeurs maîtres et des processeurs utilisateurs. Le contrôle des processeurs utilisateurs par les processeurs système requiert des voies d'accès qui ne soient pas tributaires d'autres processeurs usagers. A cet effet, un bus système relie tous les nœuds du V256, et permet de lire l'état de chaque processeur et d'effectuer des mesures de comportement.

Le projet Victor exprime plusieurs conclusions intéressantes. Il fait apparaître la nécessité d'un support matériel dédié à la communication entre processeurs. En effet, ses concepteurs estiment que 37% de la puissance de calcul des processeurs de la machine sont consommée dans l'acheminement de messages. D'autre part, un réseau d'interconnexion dédié au support système d'observation, de déverminage et de gestion d'erreurs est d'une très grande utilité.

Cette seconde proposition semble contradictoire avec la conception de machines comportant un nombre très élevé de processeurs ($\geq$ 1000). En effet, si un seul processeur en est le maître, il constitue un goulot d'étranglement certain. Certaines machines, dont Supernode, proposent un tel réseau, mais découpé selon le partitionnement de la machine.

2.6 L'architecture Supernode

Nous allons présenter maintenant une architecture d'ordinateur massivement parallèle qui a été définie et réalisée dans le projet **ESPRIT SUPERNODE I** demarré en 1986 et auquel notre équipe a participé. Nous n'examinons que les caractéristiques remarquables de l'architecture ; une étude complète est présentée par Ph. Waille dans [WM90] et [Waille91].

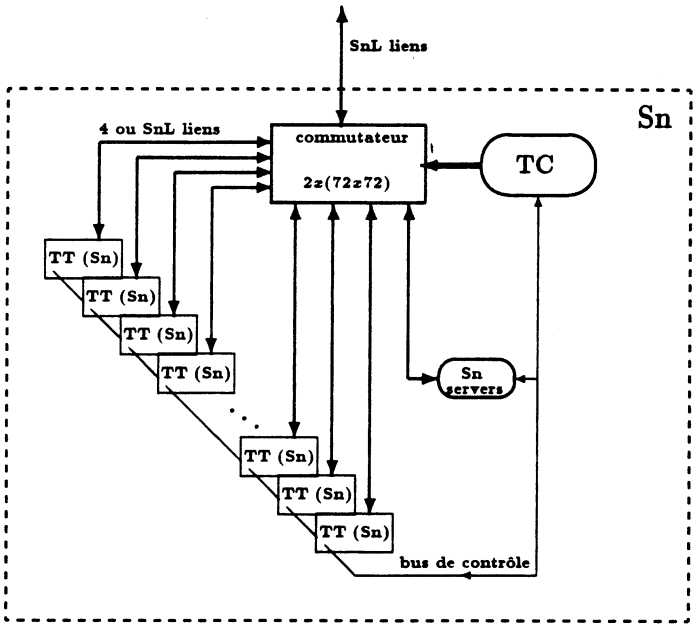


Figure 2.5: Architecture Supernode.

Le Supernode est une architecture multi-processeurs sans mémoire commune modulaire et hiérarchique ; un module de base (Supernode ou nœud de base, voir la figure 2.5) peut contenir jusqu'à 32 processeurs T800 d'une puissance d'environ 2 Mflops chacun. Les processeurs d'un module sont interconnectés à travers un commutateur programmable. Un certain nombre des liens de communication peuvent, éventuellement, être utilisés pour des connexions vers l'extérieur du module. Un tel module peut être ainsi considéré comme un *super-transputer*, avec un degré de parallélisme interne accru et un nombre adaptable de liens (SnL liens dans la figure 2.5) de communication avec son environnement, qui devient la brique de base d'une machine beaucoup plus puissante construite par répétition du même principe. Cette architecture permet d'obtenir des machines allant de la dizaine au millier de Mflops ; elle répond assez bien à notre conception de parallélisme massif.

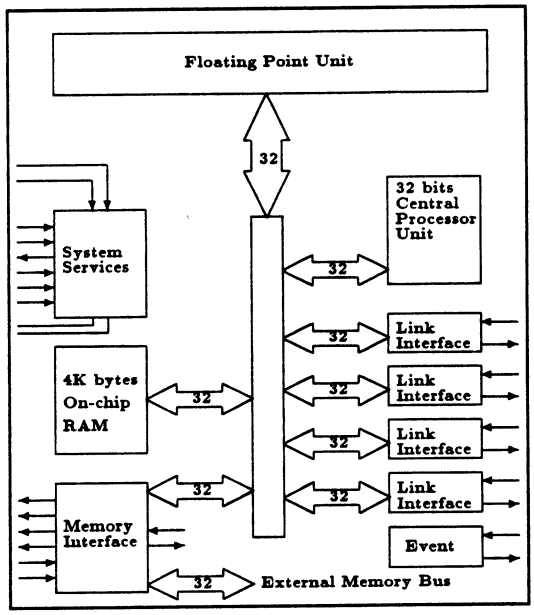


Figure 2.6: Architecture du transputer T800.

Structure d'un nœud

Parmi les éléments de la famille des transputers, le Supernode utilise comme processeurs de travail le T800 dont l'architecture est dans la figure 2.6 ; il est constitué de :

- un processeur central 32 bits.

- 4 Koctets de mémoire interne à accès rapide avec un débit de 120 Moctets/s.
- quatre liens de communication (série, bidirectionnels) à 20 Mbits/s chacun.
- une interface mémoire pouvant adresser jusqu'à 4 Gigaoctets avec un débit de 40 Moctets/s.
- un processeur de calcul flottant 64 bits permettant une performance soutenue de l'ordre de 2.25Mflops dans sa version à 30Mhz.

Le transputer, à la différence des processeurs utilisés par certains ordinateurs parallèles dont l'iPSC, le Cosmic Cube et le RP3, est un processeur conçu pour construire des architectures multi-processeurs sans mémoire commune : ses quatre liens de communication permettent de réaliser aisément des réseaux de transputers : il suffit de "souder" un lien d'un transputer à un lien d'un autre transputer pour leur permettre de communiquer, il n'y a aucun besoin de dispositifs spéciaux.

Le transputer est conçu aussi pour le parallélisme massif : il possède un noyau exécutif micro-codé permettant de simuler l'exécution parallèle d'un nombre arbitraire de processus. L'ordonnancement, se faisant par matériel, est très efficace, une commutation de contexte ne prend que quelques microsecondes. Du point de vue de la programmation, rien ne différencie les canaux logiques, simulés par l'unité centrale, des canaux physiques, implantés sur les liens de communication. De même, rien ne différencie le parallélisme logique où les processus sont exécutés sur le même transputer, que nous appelons *pseudo-parallélisme*, du parallélisme réel lorsque les processus s'exécutent sur des transputers distincts.

Le réseau d'interconnexion

C'est sans doute la nouveauté introduite par l'architecture Supernode. Le réseau d'un module de base est un Crossbar à 72 entrées et 72 sorties, l'utilisation de deux réseaux permet de construire un Supernode de base ayant jusqu'à 32 transputers de travail (TT), quelques transputers de Service (TS : disque, mémoire, etc.), et un Transputer de Contrôle (TC) qui commande le commutateur. Un bus de contrôle, transportant quelques signaux de service tels que **`reset`**, **`analyse`** et **`erreur`**, permet au TC de surveiller et contrôler, en mode maître, l'activité des TT et des TS ; ceux-ci étant les esclaves.

La reconfigurabilité du Supernode permet d'obtenir n'importe quelle topologie du réseau, même pendant l'exécution d'une application. Des topologies statiques, comme celles mentionnées dans la section 1.3.1, deviennent ainsi des cas particuliers de configuration d'un Supernode. Par ce fait, le Supernode est adaptable

aux besoins de communication des applications ; cependant, le contrôle de la communication entre processeurs devient plus délicat et plus complexe ; prenons comme exemple le routage des messages : dans un réseau statique, la stratégie de routage peut être optimisée en fonction des caractéristiques de la structure, comme c'est le cas typique des grilles et des hypercubes. Par contre, l'architecture Supernode requiert des stratégies s'adaptant à son caractère polymorphe.

Modes d'utilisation de la reconfiguration

Il y a trois types différents de reconfiguration qui peuvent être exploités dans la machine Supernode : la *reconfiguration statique*, la *reconfiguration dynamique synchrone* et la *reconfiguration dynamique asynchrone*.

Reconfiguration statique

La topologie du réseau de processeurs est fixée avant l'exécution du programme utilisateur. Aucune modification du réseau n'aura lieu pendant le déroulement du programme. Dans son fondement, ce type de fonctionnement est identique à celui des machines à topologie fixe ; la différence étant que la topologie peut être choisie à volonté, pour chaque application, avant utilisation.

Bien que la machine soit présentée comme un réseau statique, ce genre de reconfiguration fournit :

- un moyen pour obtenir la topologie de la machine qui correspond au mieux au parallélisme exploité par les programmes d'application. Dans les limites imposées par le nombre de transputers et des liens par transputer, la machine peut offrir au programme utilisateur un nombre optimum de processeurs et la topologie qui s'adapte le mieux aux structures de communication définies dans le programme.

- un moyen pour exécuter la même application sur des machines ayant des topologies différentes. Pour ce faire, il suffit de modifier la topologie du réseau avant chaque exécution de l'application. Des simulations et des études de performance des algorithmes parallèles, en fonction de la taille et de la topologie du réseau cible, sont ainsi aisément réalisables.

Dans une machine ainsi configurée, le routage des messages est nécessaire lorsque deux composantes d'un même programme, qui ont besoin de communiquer, ne sont pas exécutées sur des processeurs voisins.

Reconfiguration dynamique

Dans la reconfiguration statique, l'entité (par exemple le système d'exploitation de la machine) qui prend la décision et exécute les opération nécessaires à une reconfiguration n'a pas d'interaction avec l'application pendant son exécution ; du point de vue de la gestion des communications, le système n'offre que le routage des messages.

Nous considérons que la reconfiguration est dynamique lorsqu'elle peut avoir lieu pendant l'exécution d'une (ou plusieurs) application(s), autrement dit, l'entité qui contrôle la reconfiguration interagit avec les programmes utilisateurs. Nous décrivons ci-dessous deux sortes de reconfiguration dynamiques que nous appelons synchrone et asynchrone.

Reconfiguration dynamique synchrone

Supposons qu'un programme puisse être décomposé en plusieurs phases, chacune s'exécutant plus efficacement sur une topologie particulière du réseau d'interconnexion. Chaque phase du programme peut alors requérir une configuration différente de la machine. Des *points de reconfiguration et synchronisation* doivent être introduits dans le programme pour indiquer les changements de phase. Pour chacun des points on réalise une nouvelle configuration de la machine.

Entre deux points de reconfiguration on retrouve une configuration statique. Si le routage des messages y est utilisé, outre la configuration physique de la machine, il faudra peut être modifier aussi les routes, en les adaptant à la nouvelle configuration.

Reconfiguration dynamique asynchrone

La reconfiguration dynamique est asynchrone lorsque les demandes de reconfiguration peuvent se produire à n'importe quel moment de l'exécution d'un programme. Les demandes dépendent des besoins de communication du programme qui ne peuvent être déterminés à l'avance. La machine Supernode devrait fournir une connexion directe entre deux processeurs qui demandent de communiquer, produisant ainsi l'effet d'une machine à réseau d'interconnexion complètement connecté.

Par ce mode de reconfiguration, on pourrait penser à éliminer le concept de topologie d'interconnexion ; elle n'aurait plus de sens et le routage des messages serait remplacé par des ordres d'établissement des connexions directes entre processeurs. Tous les liens seraient libres et à l'attente d'être connectés puis

déconnectés et cela dynamiquement. Il semble cependant difficile à démontrer qu'un tel mode de fonctionnement soit plus performant que le routage des messages dans toutes circonstances : les liens deviennent une ressource partagée dont le contrôle d'accès est complexe, notamment lorsque le réseau sous-jacent, permettant d'interconnecter les liens, est bloquant ; c'est le cas général des réseaux réalisant un nombre important de points d'interconnexion.

Dans la mise en œuvre du Supernode, le réseau d'interconnexion est bloquant, bien que réarrangeable, et à contrôle centralisé sur un transputer. Toute requête de reconfiguration doit être adressée au transputer de contrôle ; il est alors le seul à décider de la validité et la réalisabilité d'une reconfiguration, qui sont affectées essentiellement par les possibles conflits de bloquage dans le réseau d'interconnexion. Dans [MuSa91] on trouve une étude des algorithmes et leur complexité.

Pour que la reconfiguration asynchrone puisse complètement remplacer le routage des messages, on doit encore beaucoup progresser dans la conception et l'efficacité des réseaux de commutation VLSI ; notamment dans leur contrôle, qui devrait permettre des décisions réparties avec la possibilité de contrôle localisé.

2.7 Conclusions

L'objectif de cette première partie était de passer en revue l'état de l'art en ce qui concerne les architectures d'ordinateurs parallèles. Cette étude nous a permis de constater que :

- Les architectures massivement parallèles permettent d'atteindre des puissances de calcul supérieures à celles des superordinateurs dits classiques.
- Les nouvelles architectures sont modulaires, ce qui leur permet de s'adapter aux utilisateurs "ayant besoin de puissances moyennes et d'un budget réduit", pour lesquels les superordinateurs sont hors de portée budgétaire.
- La diversité des approches architecturales semble se stabiliser mais, par contre, chaque approche a ses partisans et ses détracteurs dans le milieu scientifique, un consensus sur la meilleure solution ne s'est pas encore manifesté ; il est plutôt accepté que chaque approche a ses propres champs d'application, bien que ces champs se superposent [ICOT88, Tilb89, Bryan89].
- Il existe un grand nombre d'architectures expérimentales, telles que le RP3, dont l'objetif est de faire la lumière sur plusieurs problèmes encore méconnus ; elles serviront peut être à nous en donner une réponse. Pour l'instant, on avance dans deux axes principaux : d'une part se trouve la technologie qui fournit des moyens de communication et des processeurs de plus en plus performants et d'autre part le développement des modèles de programmation et d'exécution pouvant extraire les meilleures performances des architectures.

Le Supernode propose de résoudre le problème de la communication entre processeurs par deux techniques utilisées de façon complémentaire : routage des messages et reconfiguration du réseau d'interconnexion. Les performances actuelles des composants de reconfiguration, ainsi que le fait penser le réseau multi-étage utilisé dans le projet RP3, permettraient de se passer du routage des messages mais le contrôle dynamique de la reconfiguration dégrade la performance du dispositif de façon importante. De plus, les protocoles de communication ne se satisfont pas, pour des raisons des performances, de l'utilisation exhaustive de la reconfiguration. Les machines parallèles futures offriront très certainement, comme le proposent aujourd'hui Supernode et d'autres machines, les deux techniques simultanément.

Partie II

Les systèmes d'exploitation parallèles

Trois chapitres composent cette partie :

Le chapitre 3 aborde les concepts nécessaires à l'exploitation du parallélisme massif. L'objectif est de mettre en évidence le besoin de nouvelles entités de contrôle de l'exécution des programmes parallèles composés de processus communicants. Les modèles d'interaction entre les entités sont aussi examinés.

Le chapitre 4 est consacré à la présentation de Parx *: un noyau de système qui est conçu autour des modèles d'exécution et de communication appropriés au parallélisme massif. Parx est le centre de toutes nos études ; plusieurs thèmes de recherche se développent autour de lui, nous ne présentons ici que ses caractéristiques principales.*

Dans le chapitre 5, nous examinons l'état de l'art à travers quelques-uns des systèmes d'exploitation les plus connus à présent et nous comparons, autant que possible, leurs caractéristiques à celles de Parx.

Chapitre 3

Les aspects conceptuels

3.1 Introduction

La conception du logiciel de base d'un ordinateur parallèle consiste à produire un environnement d'exécution adapté aux besoins des modèles de programmation visés. L'environnement d'exécution réalise un modèle d'exécution, couramment offert par le système d'exploitation, qui doit fournir des entités compatibles à celles définies par les modèles de programmation supportés. Il doit aussi fournir les opérations de base qui découlent des politiques de gestion définies par le modèle de programmation sur les entités.

Les modèles de programmation sont reflétés par des constructeurs définis dans les langages ; ceux-ci permettent essentiellement de manipuler des structures de données et des entités qui, actives au moment de l'exécution, nécessitent un contrôle correct et efficace de la part du système d'exploitation.

Dans la démarche vers la définition d'un modèle d'exécution pour un système d'exploitation, nous examinons d'abord les formes dont le parallélisme peut être exploité, puis les paradigmes de programmation et les modes d'expression du parallélisme dans les langages. A partir de là, nous proposons des entités adaptées à la meilleure gestion de l'exécution des programmes parallèles sur des architectures multi-processeurs sans mémoire commune. Nous examinons les concepts de parallélisme et concurrence en relation aux problèmes d'ordonnancement des processus. Ensuite nous examinons les modèles d'interaction entre les entités parallèles.

3.2 Paradigmes de programmation parallèle

Le parallélisme consiste essentiellement à décomposer un problème complexe en plusieurs sous-problèmes dans le but de les traiter simultanément. La décomposition peut porter aussi bien sur les données que sur les traitements, une combinaison des deux est le plus souvent adoptée. Dans tous les cas, on construit un programme qui comporte un ensemble de traitements qui sont appliqués en parallèle sur un ensemble de données. Lors de l'exécution du programme, les traitements peuvent présenter divers degrés d'interaction car normalement ils coopèrent à la résolution d'un même problème.

Nous examinons ici les paradigmes de base de la programmation parallèle. Ces paradigmes proposent un cadre conceptuel des formes d'exploitation du parallélisme. Deux approches sont considérées dans les paragraphes suivants.

Parallélismes réplicatif, géométrique et algorithmique

Pritchard [Pritch87] et Muntean [Munt88], proposent trois formes d'exploitation du parallélisme : les parallélismes *réplicatif*, *géométrique* et *algorithmique*. Dans le premier, applicable à de nombreux problèmes scientifiques, le même traitement est répliqué autant de fois qu'il faut l'appliquer sur des données (initialement) différentes, les traitements sont plutôt indépendants les uns des autres ; ils peuvent être menés en parallèle par une "ferme" de processeurs qui communiqueront rarement.

Le second paradigme utilise la structure géométrique régulière du problème. Les traitements sont plus ou moins différents, mais chacun est responsable d'un sous-ensemble de données. Un problème est décomposé, suivant sa géométrie, en portions ne dépendant de leurs voisines que dans une sphère limitée. Les processeurs ne communiquent qu'avec ceux se trouvant dans leur voisinage immédiat.

Le paradigme du parallélisme algorithmique autorise une granularité plus fine, le problème est décomposé de telle façon que chaque traitement correspond à une petite partie du travail. Les données initiales et les résultats partiels sont échangés entre les divers traitements sans suivre un schéma nécessairement régulier. La localité des données n'est plus une caractéristique exploitable ; les communications sont alors plus complexes car la localisation, la fréquence et la durée des échanges des données peuvent varier de façon importante pendant l'exécution du programme.

Parallélismes résultat, spécialisé et agenda

Plus récemment, Carriero et Gelernter[CaGel89p] proposent eux-aussi trois formes de parallélisme, lesquelles se rapprochent des trois types examinés ci-dessus, il s'agit des parallélismes *résultat*, *spécialisé* et *agenda*.

Dans le premier, le parallélisme s'exprime en fonction de la forme du résultat attendu, qui comporte normalement plusieurs composantes décrites par des structures de données appropriées. Chaque composante du résultat est sous la responsabilité d'un traitement, en réalité, chaque traitement est encapsulé à l'intérieur d'un objet contenant les données dont il est le responsable. Tous les traitements sont activés simultanément et peuvent se dérouler en parallèle dans les limites imposées par les interdépendances possibles. Les communications ne sont pas réalisées explicitement par des opérations du genre "envoyer message" et "recevoir message" : lorsqu'un traitement a besoin de consulter les données d'un autre, il doit lire directement l'objet correspondant.

Dans le parallélisme spécialisé, les traitements sont normalement différents mais chacun est spécialisé (il ne sait résoudre qu'une seule partie du problème). Le principe s'applique aussi bien à une chaîne de production (pipeline) qu'à des problèmes où un ensemble "d'experts coopérants" est organisé dans un réseau logique ou graphe d'interactions quelconque. Le parallélisme est obtenu en activant tous les traitements simultanément. La structure du programme parallèle est déterminée par le nombre de traitements et leurs interactions, contrairement au parallélisme résultat, les données sont encapsulées à l'intérieur des traitements et la communication est réalisée explicitement par échange de messages. Il n'y a pas d'hypothèse quant aux protocoles de communication, ils dépendent des besoins particuliers du problème, par exemple, lorsque plusieurs messages peuvent converger simultanément vers un seul traitement, un protocole de réception par file d'attente est nécessaire.

Le parallélisme agenda correspond à une planification d'activités, chaque activité peut comporter plusieurs traitements. Un même traitement peut se retrouver plus d'une fois dans la même activité aussi bien que sur des activités différentes. La localisation des traitements est indépendante de celle des données, ils sont autonomes. C'est la base des structures de données distribuées.

Degré de parallélisme et granularité

Quelque soit le paradigme utilisé, le résultat de la décomposition du problème est un ensemble de traitements dont la cardinalité représente le *degré de parallélisme* que l'on a pu extraire du problème global. Le rapport de la durée moyenne d'une

séquence de traitements à la durée moyenne d'une communication est la granularité ; elle exprime le degré d'importance relative des communications dans le temps de résolution du problème.

Il est, bien sûr, de la responsabilité des outils utilisés pour réaliser la décomposition d'obtenir le degré et la granularité les mieux adaptés à la résolution du problème ; il est, par contre, de la responsabilité des langages et des fonctions d'exploitation de la machine d'offrir des moyens appropriés d'expression du parallélisme et des temps de résolution optimaux.

3.3 Modes d'expression du parallélisme

Le langage représente un modèle de programmation qui est défini par les entités manipulées par un ensemble d'opérations primitives. Certains langages offrent des constructeurs qui permettent au programmeur de désigner explicitement les entités qu'il veut faire exécuter en parallèle ; d'autres, par contre, n'offrent pas cette possibilité mais le parallélisme est implicite dans les opérations primitives d'où il est extrait par des outils spécialisés de parallélisation. Le parallélisme implicite est généralement détecté par des opérations simultanées sur une même donnée de structure complexe, ou bien par l'indépendance des opérations portant sur des données différentes.

Il existe aujourd'hui une grande diversité de langages parallèles, dans [BSTan89] on trouve un excellent état de l'art ; plus d'une centaine de langages y sont répertoriés et des entités utilisées pour exprimer le parallélisme y sont analysées, notamment il distingue les *processus*, les *objets* et les *constructeurs* comme formes d'expression explicite et les *expressions* et les *clauses* comme formes d'expression implicite. Nous considérons essentiellement le modèle de programmation du langage CSP[1] proposé par C. A. R. Hoare en 1978 [Hoare78, Hoare85] dans lequel l'entité de base est appelée processus et correspond à un flot de traitement séquentiel ayant ses propres données privées. Un certain nombre de processus, explicitement identifiés, peuvent être combinés de diverses manières, et notamment en parallèle, à l'aide des constructeurs du langage. Les processus s'exécutant en parallèle ne peuvent communiquer que par échange de messages à travers des canaux point à point unidirectionnels.

Le modèle CSP reflète directement une architecture parallèle dans laquelle une collection de processeurs sont reliés par un réseau d'interconnexion, c'est-à-dire une architecture multiprocesseurs sans mémoire commune. Aussi, le parallélisme

[1]Pour *Communicating Sequential Processes*.

spécialisé de Carriero et Gelernter s'assimile bien au modèle de programmation CSP ; cependant, dans CSP le protocole d'échange de données entre processus est bien précis : il s'agit du protocole synchrone point à point.

Plusieurs langages ont été réalisés à partir du modèle CSP, parmi eux, et pour mieux illustrer les réalisations sur des machines à base de transputers (cibles de Parx), nous considérons le langage occam [Occam2]. Dans occam, les opérations primitives sur les processus sont des constructeurs permettant de combiner des processus simples pour en construire d'autres plus complexes ; les constructeurs les plus remarquables sont **PAR** et **ALT** qui permettent d'exprimer le parallélisme et le non-déterminisme. Les opérations primitives sur les canaux sont l'envoi d'un message (noté : `canal ! expression`) et la réception d'un message (noté : `canal ? variable`). Une des caractéristiques notables du protocole synchrone est qu'il s'agit en effet d'une double opération : synchronisation et communication.

Le parallélisme exprimé dans les langages n'est que "logique" et seulement potentiel, ce n'est que lors de l'exécution du programme, et plus précisément lors de l'allocation des processus aux processeurs physiques, que l'on détermine le parallélisme réel. Par exemple, un programme comportant plusieurs processus "parallèles" peut être exécuté aussi bien sur un seul processeur que sur une architecture multiprocesseurs. Dans le premier cas il ne s'agit pas d'exécution en *parallélisme vrai* car tous les processus parallèles du programme "partagent" la même unité centrale, on dit que les processus s'exécutent en *pseudo-parallélisme*, et qu'ils sont *concurrents*.

Le parallélisme vrai entre processus ne s'obtient alors que s'il existe une correspondance bi-univoque entre l'ensemble de processus du programme parallèle et l'ensemble de processeurs physiques, c'est-à-dire que chaque processeur physique n'exécute qu'un seul processus et un seul.

Lorsque le nombre (P) de processus parallèles définis par le programme est supérieur au nombre (N) de processeurs, certains processus seront nécessairement exécutés en pseudo-parallélisme ; le parallélisme réel obtenu à l'exécution est alors égal au nombre de processseurs. Selon [Black90], un programme présente un parallélisme de N et une concurrence de P.

3.4 Gestion des entités parallèles par le système

Le *modèle d'exécution* offert par un système d'exploitation est défini par l'ensemble d'entités et l'ensemble d'opérations qui leur sont applicables. Le modèle correspond à la sémantique des entités et des opérateurs fournis par le système, qui sont eux-mêmes une virtualisation des mécanismes de base offerts par le matériel. Le système d'exploitation réalise donc un ou plusieurs niveaux de machines virtuelles et doit offrir des environnements d'exécution sur lesquels les modèles de programmation puissent se projeter efficacement.

Pour l'exécution correcte et efficace d'un programme parallèle qui peut être décomposé en processus séquentiels communicants, le système doit offrir des mécanismes pour :

1. Le placement des processus sur les processeurs.
2. Le contrôle de l'exécution des processus, c'est-à-dire la politique de gestion qui leur est appliquée, et notamment leur ordonnancement.
3. Coordonner les interactions des processus entre eux et avec leur environnement. Normalement, il s'agit des mécanismes de communication et synchronisation.
4. L'utilisation des ressources du système, notamment le support des services système usuels (fichiers, terminaux, etc.).

Le placement de processus sur les processeurs est un problème complexe qui génère un axe de recherche intéressant où les problèmes à résoudre sont encore nombreux ; ils sont étudiés au sein de l'équipe SYMPA par E. G. Talbi [TalMun90]. Des travaux intéressants sont aussi présentés dans [BCS89, AnPa88].

3.4.1 Ordonnancement des processus : parallélisme et concurrence

Les mécanismes de contrôle de l'exécution doivent fournir un environnement adéquat à l'exécution des programmes présentant divers degrés de parallélisme. Pour l'exécution d'un programme parallèle, les processus concurrents peuvent être regroupés dans des *processeurs virtuels* ; chaque processeur virtuel représentant, en principe, une unité d'ordonnancement pour le système. On a alors deux types de concurrence, celle introduite par l'ensemble de processus définis par le programme et celle introduite par l'ensemble de processeurs virtuels définis par le système. La combinaisons de ces deux types conduit, selon [Black90] à quatre types de concurrence :

1. *parallélisme pur :* chaque processus concurrent du programme utilisateur est encapsulé par un processeur virtuel et le système d'exploitation implante chaque processeur virtuel sur un processeur physique.

2. *concurrence au niveau utilisateur :* un processeur virtuel encapsule plusieurs processus concurrents et un seul processeur virtuel est implanté sur chaque processeur physique.

3. *concurrence au niveau système :* un processeur virtuel encapsule un seul processus concurrent utilisateur (il existe alors un processeur virtuel par processus concurrent) et plusieurs processeurs virtuels sont implantés sur un seul processeur physique.

4. *concurrence double imbriquée :* un processeur virtuel encapsule plusieurs processeurs concurrents (concurrence utilisateur) et plusieurs processeurs virtuels sont implantés sur un seul processeur physique (concurrence système).

3.4.2 Les entités nécessaires à la gestion du parallélisme

Chaque type de concurrence requiert du système un type de gestion différent. Par exemple, dans le parallélisme pur, lorsque les processus sont placés sur les processeurs, il n'y a plus besoin d'ordonnancement ni à l'intérieur d'un processeur virtuel ni à l'intérieur d'un processeur physique ; dans ce cas, on ne requiert aucune autre entité d'exécution sauf le processus lui-même, défini comme étant un flot de contrôle d'exécution séquentiel. Cependant, lorsque le programme utilise les ressources de la machine qui sont en dehors de son espace d'adressage, il doit faire appel à son environnement. Outre son contexte d'exécution, chaque processus doit alors contenir un ensemble d'attributs comprenant par exemple des descripteurs de fichiers utilisés et des descripteurs de l'état d'allocation de la mémoire entre autres.

L'association d'un environnement à chaque processus rend ceux-ci plus "lourds" à gérer, c'est un problème à prendre en compte, notamment lorsque l'ordonnancement imposé par la concurrence est nécessaire. Lorsqu'on veut gérer plus efficacement l'exécution des programmes à haut degré de parallélisme et qui présentent une granularité fine, il n'apparaît plus recommandable d'associer la description de l'environnement à chaque processus utilisateur. Un processus concurrent utilisateur ne devrait comporter que son contexte d'exécution ; il doit être dissocié de l'environnement. Un tel processus est alors plus "léger" à gérer notamment lors des commutations de contexte.

L'environnement peut être dissocié d'un processus mais il ne peut pas être éliminé, le processus ayant toujours besoin d'accéder aux ressources de la machine. La solution adoptée par la plupart de systèmes actuels est de placer tous les services système dans des *serveurs spécialisés*. Les processus utilisateurs s'adressent aux serveurs par échange de messages utilisant des objets de communication et des protocoles que nous examinons dans la section 8.3.

Le système gère alors des processus utilisateurs concurrents et légers ; il faut définir des politiques d'ordonnancement appropriées. Dans un programme parallèle il peut y avoir des processus qui sont en *collaboration* lors de la résolution d'un problème, tandis que d'autres sont en *compétition* pour la puissance de calcul. Les premiers peuvent être exécutés en parallélisme vrai, les seconds, par contre, seront exécutés en pseudo-parallélisme car ils sont concurrents au niveau utilisateur.

Une entité qui encapsule plusieurs processus légers concurrents, en leur donnant un environnement d'exécution avec un espace d'adressage commun est alors nécessaire ; cette entité correspond à un processeur virtuel. Aucune hypothèse n'est faite sur la façon dont ce processeur virtuel est projeté sur les processeurs physiques. Cette entité, en tant qu'abstraction, reçoit le nom de *tâche* dans les systèmes Mach [TeRa87], Pcl [IsBor90] et Parx ; dans le système Chorus [AGHR89] on l'appele *acteur* et dans Peace [WSP90] il s'agit d'un *team*.

Les processus légers (couramment appelés "*threads*" dans la littérature) et les *tâches* ne sont pas suffisants pour supporter l'exécution d'un programme parallèle, il est encore nécessaire de permettre l'exécution en parallélisme vrai, et en collaboration, de toutes les *tâches* du programme, c'est-à-dire qu'il faut établir une politique globale cohérente de l'exécution du programme sur un ensemble de processeurs physiques. Une autre entité, qui porte maintenant sur tout le programme utilisateur, est donc nécessaire. Cette entité sert au système pour protéger les espaces d'adressage des programmes différents et pour réaliser la comptabilité des ressources utilisées par les programmes.

Une telle entité n'est pas toujours définie d'une façon explicite par les systèmes d'exploitation. Dans le cas du système Mach, le besoin n'a été ressenti que par la demande des utilisateurs qui désiraient utiliser le parallélisme mais n'en avaient pas les moyens avec le modèle de *tâches* et *threads* offert par Mach. La possibilité de coopération entre *tâches* s'exécutant en parallèle n'a été rajoutée que récemment par des librairies qui réalisent des fonctions du genre *multi-fork*. Cependant, une telle démarche ne reflète pas une politique cohérente de gestion du parallélisme. En ce que concerne Chorus, la coopération entre différents ac-

teurs peut se faire en les regroupant autour des *ports* de communication ; une entité spécifique de gestion du parallélisme n'existe pas.

Les systèmes plus orientés vers le parallélisme, tels que Peace et Parx, considèrent dès leur conception une entité de gestion du parallélisme. Dans le premier on l'appelle *League* et dans le second nous l'appelons *Ptâches* (pour *tâches* parallèles ou "Parallel tasks").

Les approches "système parallèle" illustrée ici par Parx, Peace et Pcl, et "système réparti", répresentée ici par Mach et Chorus, diffèrent par le fait que la première prend en compte la problématique concurrence-parallélisme comme un tout auquel il faut donner une solution cohérente dès le départ. Les deux approches coïncident dans une partie du modèle : *threads* et *tâches* ont un équivalent dans chaque système. Un consensus semble se dégager en ce que concerne le besoin d'une entité telle que la *Ptâche* de Parx ; tous les systèmes convergent, par des voies différentes, vers la définition d'une entité équivalente de gestion globale du parallélisme. Les divers systèmes mentionnés ici seront examinés avec plus de détails dans le chapitre 5.

3.5 Modèles d'interaction entre entités parallèles

Des modèles de communication et synchronisation sont nécessaires pour coordonner les interactions entre processus[2]. Il existe essentiellement deux modèles de base : la mémoire partagée et l'échange de messages. Dans le premier, les processus interagissent par la voie d'un espace d'adressage global, partagé par tous les processus. Dans le second, les processus interagissent par l'intermédiaire d'objets de communication partagés. A la différence d'un espace d'adressage, un objet de communication n'est partagé que par les processus qu'il met en correspondance lors des communications.

Nous sommes essentiellement concernés par le modèle basé sur l'échange de messages, cependant, il est intéressant d'examiner succinctement les tendances actuelles basées sur le modèle de mémoire partagée.

[2]Bien que nous ayons donné des noms différents à des entités jouant un rôle différent dans le modèle d'exécution, lorsqu'un concept ou une technique s'applique à toutes les entités, nous allons utiliser le mot "processus" comme un nom générique.

3.5.1 Interaction par données partagées

Les techniques d'implantation de la mémoire partagée conduisent essentiellement à deux mécanismes de base pour l'interaction par des données partagées. Dans le premier, les données sont universellement manipulables ; un ensemble d'opérations, exclusives ou non, sont définies pour accéder à chaque donnée partagée, ces opérations entraînent une définition du nommage, contextuel ou associatif, des données, et des mécanismes de synchronisation entre protagonistes. Ces derniers sont anonymes dans cette communication.

Dans le second, les données sont affectées une fois pour toutes. Chaque variable partagée n'est affectée qu'une seule fois et possède un état : initialisée ou non. Les opérations d'accès peuvent se synchroniser en fonction de l'état de la variable. Chaque entité active possède un ensemble de droits (lecture, écriture, etc.) par rapport aux données partagées. On peut enrichir cette technique plutôt rigide : il est possible d'affecter à une variable partagée une structure comprenant des valeurs et des variables. Ces dernières pourront à leur tour être affectées par un autre processus qui laissera des variables pour la suite de la communication. Concurrent Prolog [Shapiro86] est un exemple de ce modèle.

Le langage Linda [CaGel89l] est représentatif du premier paradigme ; il propose un modèle de communication et synchronisation par échanges de *n-uplets* contenus dans une mémoire associative commune : l'espace des *tuples.* La communication s'effectue à travers un espace *commun* de capacité théoriquement infinie, la correspondance entre requêtes s'effectue par "pattern matching". La donnée d'un n-uplet, dont certains éléments sont des valeurs et d'autres sont des variables, permet de retrouver un n-uplet ayant les mêmes valeurs pour les mêmes éléments et d'obtenir la valeur de ses variables. La synchronisation se fait par l'atomicité des opérations de lecture, qui peuvent être bloquantes (synchrones) ou non bloquantes (asynchrones), et d'écriture, toujours synchrones. Le modèle d'interaction entre processus est appelé "generative communication" [Geler85] : les n-uplets peuvent être actifs, on parle alors de processus, ou passifs, ce sont alors des données ; pendant qu'ils sont actifs, les processus sont dans l'espace des tuples, ils redeviennent des données ordinaires à leur terminaison.

3.5.2 Interaction par échange de messages

Dans le modèle de processus communicants, l'interaction entre processus passe par des objets de communication partagés par les processus qui communiquent. Ces objets peuvent prendre des formes diverses et peuvent supporter divers protocoles de communication et de synchronisation. Les objets permettent l'échange des messages et les protocoles assurent un type d'échange dans des règles précises,

ceux-ci définissent alors la sémantique de la communication.

L'unité atomique d'échange d'information entre processus est le *message* ; il correspond à une suite d'octets –ou des bits– et peut être plus ou moins structuré selon les besoins de contrôle de son acheminement et de son interprétation par les protocoles. La taille d'un message est généralement variable mais bornée. Il contient souvent deux parties : un *en-tête* et un *corps*. L'*en-tête* contient des informations nécessaires au contrôle de la communication, par exemple : identification des processus émetteur et récepteur du message, identification du protocole à suivre et longueur du *corps* entre autres. Le *corps* contient les données transférées par le message.

Objets et protocoles de communication

Un *canal* permet la communication point à point entre deux processus ; l'un des processus est l'émetteur et l'autre le récepteur. Un canal est unidirectionnel lorsque les rôles d'émetteur et de récepteur ne peuvent être intervertis. Des protocoles différents peuvent être définis entre les deux processus communicants, cependant, pour un modèle de programmation qui admet une granularité fine, les protocoles implantés dans les canaux doivent être efficaces afin que le système puisse supporter le nombre de canaux requis par les programmes ; ce nombre peut être très grand et très variable d'un programme à un autre. L'efficacité concerne notamment l'utilisation de la mémoire et les actions système consacrées au contrôle du déroulement du protocole.

Des schémas de communication et de synchronisation plus complexes, pour lesquels les canaux point à point ne sont pas bien adaptés, sont souvent nécessaires. Considérons l'exemple suivant : supposons que l'on doive contrôler l'exécution parallèle d'un ensemble de processus créé par un constructeur parallèle. Un processus père qui lance l'exécution simultanée de tous les processus requiert un objet de communication qui réalise un protocole *un vers n* ; les processus peuvent communiquer entre eux, pendant leur exécution, à travers des canaux point à point (*un vers un*). Enfin, il faut que le processus père puisse avoir connaissance de la terminaison des processus qu'il a lancé ; il doit alors recueillir un message de chaque processus signalant leur terminaison, un objet de communication qui réalise un protocole *n vers un* est alors nécessaire. Une méthodologie de construction des objets qui offrent divers schémas et protocoles de communication, appelés *connecteurs*, a été proposée par M. Riveill[Rive87]. L'idée est de construire quelques connecteurs de base à partir desquels on pourrait bâtir d'autres plus complexes.

Les canaux et les connecteurs sont des objets ayant respectivement deux et plusieurs extrémités, auxquelles les processus qui communiquent sont rattachés. Même lorsque l'ensemble de processus communicants pourrait se modifier d'une communication à une autre, l'ensemble des processus qui participent dans une communication est bien déterminé avant que celle-ci n'ait lieu. Un système d'exploitation, dans lequel les services système sont placés dans des serveurs auxquels des processus clients adressent des requêtes, requiert un objet de communication un peu différent, dans le sens où un seul processus lui est attaché : le serveur. Les processus qui ont une connaissance de l'existence de l'objet peuvent envoyer des messages, mais ils ne sont pas nécessairement attachés à l'objet. Un tel objet de communication est couramment appelé *port*. Un *port* correspond typiquement à une boîte aux lettres sur laquelle les processus peuvent réaliser certaines opérations définies par des droits d'accès, par exemple tout processeur connaissant l'adresse du *port* peut y déposer un message. Le processus auquel le *port* est attaché est le seul à pouvoir lire les messages.

Modèles de communication-synchronisation

Une communication entre processus induit nécessairement une synchronisation particulière. En règle générale, un système met en œuvre plusieurs formes de communication-synchronisation : pour chaque envoi de message, le processus émetteur est bloqué pendant un temps plus ou moins long. Voici les modèles de communication-synchronisation de base :

1. **Communication par rendez-vous** : le rendez-vous est un protocole dans lequel les processus qui participent s'attendent mutuellement (se synchronisent fortement) avant qu'aucun message ne soit transmis ; il s'agit d'un protocole synchrone qui ne requiert pas de stockage intermédiaire du message. Ce protocole est à la base de l'interaction entre processus dans le modèle CSP ; il est aussi celui supporté par le langage occam. Dans ce dernier cas, le protocole est implanté sur un canal, par conséquent, seulement deux processus, ceux connectés par un même canal, participent à un rendez-vous. Cependant, plusieurs rendez-vous peuvent avoir lieu simultanément.

   L'initiative de la communication est partagée : pour que la communication puisse avoir lieu, il est nécessaire que les deux processus soient prêts. La synchronisation permet d'assurer qu'un message émis par un processus est toujours[3] reçu par le processus récepteur. Le système n'a pas besoin de stocker les messages car ceux-ci ne sont transmis que lorsque le récepteur est prêt : les besoins de mémoire sont à la charge des processus utilisateurs.

[3]En supposant que les moyens de communication sous-jacents soient fiables.

2. **Communication asynchrone** : l'asynchronisme implique essentiellement que le processus émetteur prend seul l'initiative de la communication, sans se préoccuper de l'état du processus récepteur. L'émetteur fait l'hypothèse que la communication a lieu correctement : après avoir émis le message, il continue son exécution comme si le message avait été traité par le récepteur. Cependant, l'émetteur peut ultérieurement se mettre en attente d'une réponse du récepteur. L'émetteur n'est bloqué que le temps nécessaire au système pour prendre en compte sa demande.

   Le système doit pouvoir stocker les messages émis et non encore consommés. La notion d'asynchronisme, et de stockage des messages en réception, s'associe alors naturellement avec les *ports*. L'espace mémoire réservé par le système pour stocker les messages adressés à un *port* ne pouvant être illimité, des messages peuvent être refusés. Lors du refus d'un message, le système doit prevenir l'émetteur, celui-ci peut alors décider de renvoyer le message. Il faut éviter qu'un même message ne soit toujours refusé ; des mécanismes évitant la famine sont alors nécessaires.

3. **Appel de procédure à distance ou invocation** : l'émetteur est bloqué jusqu'à ce que le processus destinataire du message en ait effectué le traitement et ait renvoyé une réponse. Ce modèle de communication est une extension de l'appel de procédure connu dans les langages séquentiels, l'émetteur prend l'initiative de l'interaction, il envoie les paramètres de l'appel au processus distant et se met en attente de la fin d'exécution de la procédure ; il ne reprend son exécution qu'après avoir reçu les paramètres de retour.

Primitives de communication

Les primitives de base pour la communication sont souvent nommées **send** (ou `output`) pour l'envoi d'un message, et **receive** (ou `input`) pour la réception d'un message. On dit qu'une primitive **output** est bloquante lorsque le processus qui l'exécute ne peut continuer son exécution qu'après avoir été prévenu de la réception du message par le correspondant. Une primitive **input** est bloquante si le processus ne peut continuer son exécution qu'après avoir reçu un message. Le protocole de rendez-vous fait usage des primitives **output** et **input** bloquantes.

L'invocation est souvent considérée comme une primitive même si elle peut induire un protocole de communication à plusieurs phases. Comme l'émetteur peut prendre la décision de communiquer sans connaître l'état du processus destinataire, il doit exécuter une primitive **output** non bloquante suivie immédiatement par une primitive **input** bloquante.

Anonymat des processus communicants

Dans le modèle CSP, le processus émetteur d'un message doit identifier explicitement le processus récepteur ; dans occam, par contre, ceci a été modifié par l'envoi d'un message "sur un canal", le processus émetteur dépose le message sur le canal sans se soucier de l'identification du processus récepteur. Le langage occam impose cependant un attachement statique d'un canal à deux processus, c'est-à-dire qu'un canal n'a pas une existence propre en tant qu'objet de communication, il existe parce qu'il est associé aux processus qu'il connecte.

Ces deux choix conceptuels : désignation explicite du processus récepteur et allocation statique des canaux aux processus, facilitent la formulation d'un modèle et la mise en œuvre par un langage, un système d'exploitation ne peut cependant se contenter de cela. En règle générale, les *ports* et les canaux peuvent avoir une existence indépendante des processus qui les utilisent ; ils peuvent alors être attachés aux processus participant à une communication à un instant donné mais ils peuvent être relogés, favorisant ainsi la reconfiguration du système par rapport aux besoins de communication ; ils sont des objets intermédiaires de communication auxquels les messages s'adressent sans nommer explicitement les récepteurs.

Des choix d'implantation par les systèmes peuvent varier entre les deux extrêmes. Par exemple, dans le système Chorus, les *ports* peuvent migrer d'un acteur à un autre et ils peuvent être activés et désactivés dynamiquement, ce n'est pas le cas de tous les systèmes. Les systèmes Mach, Chorus, Parx et Peace ont tous un objet de communication assimilable aux *ports*. Nous verrons plus tard que la définition précise peut varier d'un système à un autre.

3.6 Conclusions

A propos de l'architecture des systèmes

Un consensus semble être acquis quant à la nécessité d'un noyau minimal gérant efficacement un nombre limité d'abstractions et surtout de ressources, notamment les processeurs et les moyens de communication. Ce noyau minimal s'exécute sur chaque nœud de la machine.

La gestion des ressources globales (mémoire secondaire par exemple) est de plus en plus reléguées à des serveurs qui se placent hors du noyau et au même niveau que l'utilisateur ; le cœur du système se contente d'offrir des mécanismes permettant de mettre en œuvre les politiques de gestion.

A propos du modèle d'exécution

La tendance est à distinguer l'espace d'adressage des flots d'exécution. L'expression du parallélisme passe par une entité qui représente un "flot d'exécution" dans un environnement donné. Plusieurs flots peuvent s'exécuter "simultanément" et les politiques d'ordonnancement doivent être bien adaptées aux environnements particuliers. Certains flots sont en compétition, d'autres sont en collaboration. Les politiques d'ordonnancement sont désormais dépendantes des besoins spécifiques à une application. Il n'est plus possible d'appliquer une même politique globale sans léser certains types d'applications [Black90]. De même que les travaux interactifs avaient révolutionné les techniques d'ordonnancement, le parallélisme et le pseudo-parallélisme viennent remettre en question ces techniques ; ils distinguent respectivement l'exécution simultanée sur des processeurs différents et sa simulation logicielle opérée par un noyau de système.

A propos de l'interaction entre processus

La communication passe par des objets intermédiaires qui permettent d'éliminer les contraintes de localisation géographique et autorisent ainsi la migration des objets.

Les protocoles de communication offrent plusieurs variantes, et il est admis que chaque classe d'applications requiert un ensemble de protocoles spécifiques. Aussi, lorsqu'on veut fournir un environnement pour l'exécution des programmes parallèles, les protocoles utilisés au niveau système deviennent inefficaces pour la communication entre processus utilisateurs ; une varieté de protocoles moins lourds est nécessaire.

Tout accès à des données non locales utilise nécessairement des mécanismes de communication de bas niveau, incluant des mécanismes physiques de transmission des données, qui sont toujours limités. Des mécanismes logiciels qui virtualisent l'utilisation des ressources physiques en donnant l'impression qu'ils sont en nombre et capacité suffisants sont nécessaires.

La mise en pratique des concepts

Dans le chapitre suivant nous allons examiner le système Parx, un noyau de système pour les ordinateurs multiprocesseurs sans mémoire commune, dont la conception s'appuie sur les modèles examinés dans ce chapitre et tient compte des remarques que nous venons de faire.

Chapitre 4

Mise en œuvre par Parx

4.1 Introduction

Dans le chapitre précédent nous avons étudié les principaux concepts impliqués dans la réalisation d'un système d'exploitation pour des architectures d'ordinateurs à haut degré de parallélisme. Ces concepts sont intégrés d'une façon cohérente par Parx : un noyau de système d'exploitation qui offre un environnement bien adapté à la programmation parallèle sous le paradigme de processus communicants.

Dans ce chapitre, nous allons examiner quelques-uns des aspects de la conception de Parx, une étude complète est présentée par Langué dans [Langue91]. Nous examinons d'abord l'approche générale adoptée pour Parx, qui consiste à fournir de machines virtuelles, lesquelles offrent aux utilisateurs la possibilité de s'abstraire plus ou moins de la machine physique. Ensuite nous présentons l'architecture de Parx et la manière dont Parx intègre le modèle d'exécution proposé dans le chapitre précédent. Enfin, nous présentons le noyau de communication qui supporte le modèle d'interaction entre processus.

4.2 Structure générale de Parx

Les objectifs généraux de Parx sont :

a) offrir un environnement permettant la programmation parallèle indépendamment du degré et du grain du parallélisme des programmes utilisateurs.

b) offrir un support système qui réalise une gestion correcte et efficace du parallélisme et des moyens de communication offerts par l'architecture physique de la machine.

c) offrir un système multi-utilisateur.

Pour le dernier objectif, le système doit permettre à différentes classes d'utilisateurs d'accéder simultanément à la machine, chacun ayant sa propre machine virtuelle, laquelle met en œuvre des niveaux d'abstraction en dessus de la machine physique.

Adaptation aux besoins des utilisateurs

L'intérêt d'offrir des niveaux d'accès à la machine s'explique par les besoins des divers utilisateurs. A un extrême, il est des utilisateurs qui veulent contrôler eux-mêmes les ressources de la machine, notamment les processeurs et la topologie du réseau d'interconnexion. Ce sont des utilisateurs qui ont besoin d'exprimer et de contrôler explicitement le parallélisme logique et physique ; par exemple, ceux qui programment actuellement en occam des réseaux de transputers voudraient accéder à la machine de cette façon pour exécuter des programmes sans modification.

A l'extrême opposé, il est des utilisateurs qui ne veulent même pas programmer en exprimant du parallélisme, mais qui veulent obtenir l'exécution la plus rapide de leurs programmes. Le système doit offrir alors des niveaux d'abstraction tels que l'utilisateur n'ait pas à contrôler le parallélisme, c'est le système qui doit l'exprimer et l'exploiter efficacement. Il est bien sûr des utilisateurs qui se placent entre les deux extrêmes.

Le système d'exploitation doit offrir un environnement d'exécution sur lequel le modèle de programmation puisse se projeter efficacement et sans être trop déformé. La facilité de cette projection rend compte aussi bien de l'adéquation réciproque des modèles d'exécution et de programmation que de la simplicité d'utilisation de l'interface fournie par le système. Les objectifs de simplicité et d'efficacité d'utilisation de la machine sont normalement contradictoires, de nombreux systèmes d'exploitation ont fait le choix de dissimuler complètement la complexité du parallélisme de la machine physique à l'interface de la machine virtuelle. Dans Parx, par contre, le choix a été d'offrir des interfaces plus souples.

Machine virtuelle

La notion de machine virtuelle permet de construire un système multi-utilisateur : le système offre à chaque utilisateur une interface identique à celle de la machine physique partagée. Dans le cas des architectures mono-processeurs, le système offre une machine virtuelle à chaque utilisateur en contrôlant le partage par multiplexage du seul processeur physique disponible. Le multiplexage consiste à allouer

le processeur aux usagers par tranches de temps. Dans sa généralité, cette notion est extensible aux systèmes parallèles.

Un programme parallèle nécessite un environnement d'exécution et un contrôle spécifique et cohérent ; pour cela, la machine virtuelle fournie par le système doit être réellement parallèle, avec un support bien adapté. Dans le cas des architectures parallèles multi-processeurs que nous considérons ici, l'ensemble des processeurs physiques peut être divisé en plusieurs groupes, chacun définissant une machine virtuelle que correspond à un sous-réseau pouvant être configuré selon les besoins du programme utilisateur s'y exécutant ; chaque machine virtuelle peut être configurée indépendamment des autres.

Une machine virtuelle offre à l'utilisateur les services système nécessaires et réalise les modèles d'exécution et de communication entre processus. Elle est donc un domaine protégé et servi par le système. A l'intérieur d'une machine virtuelle, un programme utilisateur est libre pour configurer et reconfigurer le réseau d'interconnexion. La machine globale est donc vue comme un ensemble de sous-réseaux de processeurs pouvant évoluer dynamiquement.

Des protocoles de communication appropriés permettent la communication entre processus s'exécutant sur des machines virtuelles différentes. Ainsi, une machine virtuelle "utilisateur" peut communiquer avec une machine virtuelle "serveur système" ou "sous-système". Un protocole de communication tel que client-serveur apparaît clairement nécessaire.

En résumé, la notion de machine virtuelle permet :

- d'offrir un environnement multi-utilisateurs ; chaque utilisateur a une machine virtuelle adaptée à ses besoins, protégée des autres utilisateurs et ayant l'accès aux serveurs système pour bénéficier des ressources globales : système de fichiers, mémoire secondaire, etc.
- d'offrir aux utilisateurs une quantité des processeurs bien adaptée à leurs besoins. Autrement dit, d'offrir un environnement d'exécution dynamiquement adaptable ; les machines virtuelles pouvant changer de topologie et de taille : des processeurs peuvent y être rajoutés ou libérés.
- d'offrir l'environnement d'exécution le plus efficace : les protocoles de communication entre processus qui sont strictement nécessaires et mieux adaptés aux besoins du programme.
- d'offrir un environnement intégré multi-systèmes : des sous-systèmes différents peuvent co-habiter dans une seule machine.

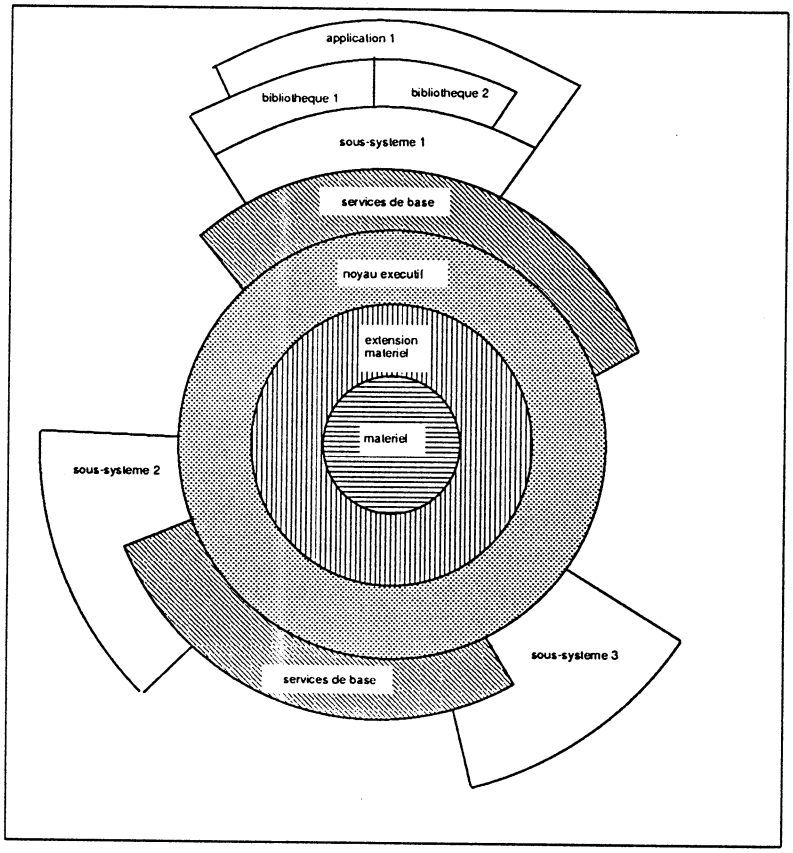


Figure 4.1: Structure générale du système Parx.

Organisation

La figure 4.1 est une illustration de la structure globale de Parx. Une première couche de logiciel étend les possibilités du matériel et nous la dénommons "extension matérielle". Elle permet le découpage dynamique de la machine physique en de multiples machines virtuelles, et met en œuvre les mécanismes de communication de base : acheminement correct des messages entre toute paire de processeurs à l'intérieur de chaque machine virtuelle et virtualisation des liens de communication. Elle a aussi pour fonction de fournir le flot d'exécution (ou processus léger) sur le processeur physique. A ce niveau, l'interface offre une machine dont les moyens de communication entre processeurs ont été virtualisés : l'extension matérielle émule un réseau d'interconnexion complètement connecté. Un utilisateur peut être aussi bien un bâtisseur de systèmes qu'un programmeur des applications qui gère lui-même la projection de ses programmes sur la machine physique.

Une seconde couche, le noyau, implante le modèle de processus, notamment leur

désignation, protection et ordonnancement, la gestion mémoire, le traitement des exceptions et les autres services système de bas niveau (horloge, pilotes de périphériques etc.). Le noyau offre un ensemble de mécanismes de contrôle des abstractions qu'il construit. Il implante aussi le modèle d'interaction entre processus par la mise en œuvre des protocoles de communication.

L'interface offerte par le noyau permet aux bâtisseurs de systèmes d'implanter les services système sous la forme de serveurs spécialisés. Les programmeurs des applications peuvent accéder au noyau lorqu'ils n'ont pas besoin des services de base mais seulement d'utiliser les mécanismes d'interaction entre processus, notamment les protocoles de communication. Au-dessus du noyau se trouvent les services de base, sous la forme de serveurs, par exemple ceux permettant les accès aux fichiers ou le chargement des programmes parallèles.

La couche suivante est subdivisée en sous-systèmes. Chacun propose à un ensemble d'utilisateurs un environnement de développement et d'exécution de programmes. Chaque sous-système peut mettre en œuvre son propre ensemble de politiques à partir des mécanismes fournis par le noyau et aussi les services de base de Parx. Chaque sous-système offre à ses utilisateurs une vue particulière de la machine, par exemple, celle d'une machine Unix standard.

4.3 Mise en œuvre de la machine virtuelle : les *clusters* de Parx

Le *cluster* est le support d'exécution d'un ensemble de *tâches* parallèles. Il comporte un ensemble de processeurs formant une configuration au gré de l'utilisateur. Le *cluster* représente la machine virtuelle allouée par le système à une application. Il est aussi un domaine de communication, et peut être configuré de manière à privilégier une topologie donnée. Il peut évoluer dynamiquement par ajout ou retrait de processeurs ou par changement de topologie.

Le domaine de protection, correspondant à un espace d'adressage dans un système d'exploitation classique, est le *cluster*. L'intégrité de chaque *cluster* vis-à-vis des autres est assurée. La manipulation du domaine d'un *cluster* s'effectue au travers d'opérations dont la légalité est vérifiée par les mécanismes de protection des entités du système.

A sa création, un nombre minimal de processeurs est requis pour le *cluster*, ainsi qu'un nombre optimal. Le premier correspond à une borne inférieure au-delà de laquelle le créateur ne peut se satisfaire de l'allocation. Le second correspond au

nombre de processeurs effectivement souhaité. Ces bornes peuvent être déterminées par l'analyse d'un algorithme, qui permet d'évaluer un nombre optimal de processeurs pour un critère de performance donné. On peut de même obtenir un nombre minimal pour un seuil de performance acceptable.

Rôle de la reconfiguration du réseau d'interconnexion

Le concept de *cluster* est suffisamment général pour être applicable à toute architecture multi-processeurs sans mémoire commune ; leur utilisation doit être cependant adaptée aux caractéristiques propres d'une machine. Ces caractéristiques, nous l'avons vu dans le chapitre 1, sont particulièrement déterminées par le réseau d'interconnexion entre processeurs. Le concept de *cluster* implique en général une dynamicité qui ne peut être totalement exploitée que si le réseau d'interconnexion est reconfigurable ; la structure d'un *cluster* peut changer par modification dynamique de :

- sa taille, qui correspond au nombre de processeurs lui appartenant,
- sa configuration interne, déterminée essentiellement par la topologie du réseau d'interconnexion entre processeurs,
- ses interconnexions avec d'autres *clusters*.

Le concept de *cluster* est néanmoins utilisable dans des machines à réseau d'interconnexion statique, mais toutes ses propriétés ne seront pas exploitables.

Dans la figure 4.2, on peut voir un schéma représentatif de l'application de la notion de *cluster* ; on y trouve trois *clusters* : un *cluster système* (séparé en deux serveurs, appelés **serveur_A** et **serveur_B**, par souci de clarté de la figure), et deux *clusters utilisateurs* appelés **application_1** et **application_2**. La reconfigurabilité joue un rôle primordial pour obtenir une organisation de machine comme celle de la figure 4.2 où l'on peut constater aussi que les deux *clusters* utilisateurs ont des topologies différentes.

La possibilité de reconfiguration du réseau d'interconnexion sert à :

- créer des *clusters*, permettant ainsi d'organiser la machine réelle en machines virtuelles par partitionnement dynamique du réseau d'interconnexion,
- configurer et reconfigurer chaque *cluster* selon les besoins de l'application qui s'exécute,
- établir dynamiquement des connexions directes entre *clusters*.

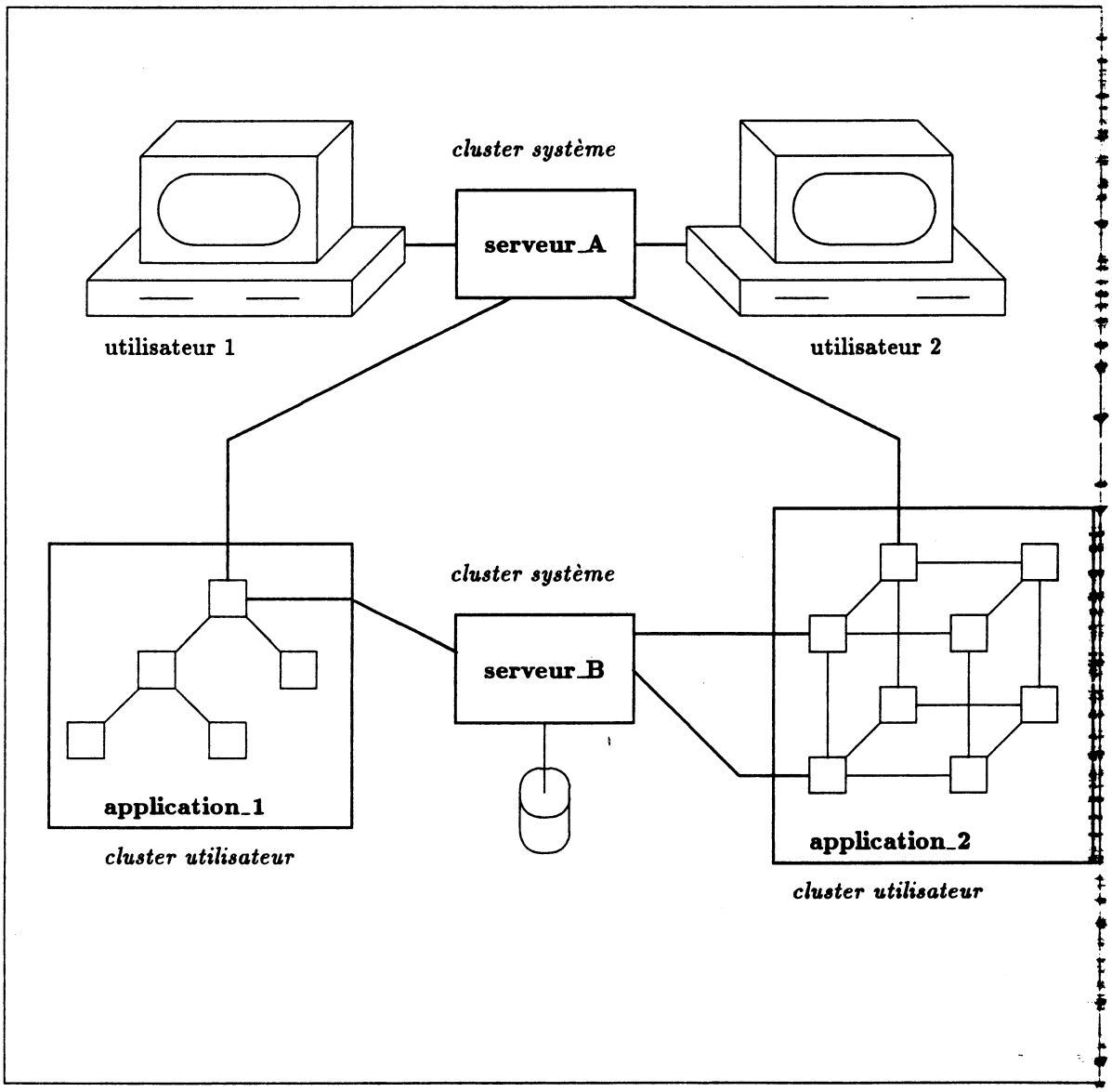


Figure 4.2: Illustration du rôle de la reconfiguration dans la création de *clusters*.

A chaque *cluster* on associe un *domaine de communication* (voir section 4.6) différent, des communications inter-*domaines de communication* sont nécessaires pour donner accès au *cluster système* et éventuellement pour permettre la collaboration entre *clusters* supportant des applications au niveau utilisateur. Les *clusters* utilisateurs correspondent en fait au mode esclave des processeurs classiques. Certains appels systèmes sont réalisés par l'envoi explicite de messages au *cluster* système. D'autres peuvent s'exécuter localement. L'ensemble du *cluster* utilisateur, possédant différents flots de contrôle s'exécutant en parallèle, n'est cependant pas suspendu, seul l'est le flot d'exécution ayant provoqué l'appel. Ceci a une conséquence importante quant à l'observabilité d'un état figé d'une machine virtuelle utilisateur : il est impossible de l'obtenir lors de chaque appel système.

Synopsis des opérations applicables aux clusters.

```
cluster_create(id_cluster, proc_type, nb_proc_min, nb_proc,
          quota, id_proc[])
cluster_delete(id_cluster)
cluster_add(id_cluster, nb_of_proc, proc_type, id_proc[])
cluster_remove(id_cluster, nb_act_proc, id_procs[])
cluster_resize(id_cluster, nouveau_quota)
cluster_set_configuration(id_cluster, nouv_config)
cluster_get_configuration(id_cluster, config)


id_cluster :  identificateur du cluster.
proc_type  :  type de processeur, par exemple T414, T800, T9000.
nb_proc_min :  borne inférieure de processeurs à allouer.
nb_proc :  nombre de processeurs ''idéalement'' souhaité.
nb_act_proc :  nombre de processeurs dans le cluster.
nb_of_proc :  nombre de processeurs à rajouter au cluster.
quota :  borne supérieure du nombre de processeurs .
nouveau_quota :  nouvelle borne supérieure.
id_proc[] :  identification des processeurs du cluster.
config :  configuration actuelle du cluster.
nouv_config :  nouvelle configuration souhaitée.
```

4.4 Le modèle de processus

Le modèle de processus de Parx est construit en trois niveaux par les entités : *thread*, *tâche* et *Ptâche*. Une *Ptâche* est une entité qui permet de manipuler le parallélisme et la distribution ; elle correspond à un programme parallèle en cours d'exécution sur une machine virtuelle correspondant aux besoins du programme : le *cluster*.

Une *Ptâche* met en œuvre le modèle de programmation visé par le programmeur ; elle est composée d'un ensemble de *tâches* s'exécutant sur des processeurs virtuels ne partageant pas de mémoire. Chaque processeur virtuel supporte une *tâche*, et ils démarrent suivant un mode de synchronisation donné. Le nombre de processeurs virtuels d'une *Ptâche* peut évoluer dynamiquement.

Une *Ptâche* est sous le contrôle du système, lequel peut réaliser un ensemble d'opérations concernant par exemple la suspension et la reprise de l'exécution ; chaque opération porte sur chacune des *tâches* appartenant à la *Ptâche* sur laquelle elle est appliquée. Plusieurs *tâches* peuvent être créées, suspendues, reprises et détruites. Ces opérations s'appliquent aux *threads* s'exécutant dans leur espace d'adressage. Différentes *Ptâches* s'exécutent dans des espaces de mémoire séparés et ne peuvent communiquer que par échange de messages sur des objets de communication protégés par le système.

Une *tâche* est l'unité d'allocation de mémoire et correspond à un espace d'adressage protégé, dans lequel peuvent s'exécuter en pseudo-parallélisme plusieurs flots de contrôle que l'on appelle *thread*. Le *thread* est l'unité élémentaire d'exécution ; il représente le grain fin d'exécution sur la machine. Un *thread* s'exécute dans le contexte d'une seule *tâche*, mais une *tâche* peut contenir plusieurs *threads*, dans ce cas ils sont, du point de vue de l'ordonnancement, en compétition pour l'usage du processeur.

Le *thread* se limite à un compteur ordinal et quelques registres lui permettant d'accéder à un espace d'adressage ; il représente le minimum d'information nécessaire à l'exécution d'un flot de contrôle séquentiel, le temps de commutation de contexte est ainsi réduit au strict minimum.

Sypnosis des opérations sur les Ptask, task et threads.

```
Ptask_create(id_cluster, id_Ptask, persistance)
Ptask_delete(id_Ptask)
Ptask_suspend(id_Ptask, nb_task, task[])
Ptask_resume(id_Ptask)
Ptask_info(id_Ptask, Ptask_info)
Ptask_identity(id_Ptask)

task_create(id_Ptask, nb_task, liste_descrip,
      liste_task, persistance)
task_delete(id_Ptask, id_task)
task_suspend(id_Ptask, id_task, nb_threads, id_thread[])
task_resume(id_Ptask, id_task)
task_information(id_Ptask, id_task, task_info)
task_identity(id_task)

thread_create(id_Ptask, id_task,id_thread[], nb_threads)
thread_delete(id_Ptask, id_task, id_thread)
thread_suspend(id_Ptask, id_task, id_thread[], nb_threads)
thread_resume(id_Ptask, id_task, id_thread[], nb_threads)
thread_identity(id_thread)


(Dans la description ci-dessous  :  Proc = Ptask ou task ou
thread)

id_Proc :  identificateur du processus.
nb_Proc :  nombre de processus impliqués dans l'opération.
Proc[] :  tableau contenant l'identification des processus.
liste_descrip :  décrit zones de mémoire, ou segments.
liste_tâches :  liste de tâches créées.
Proc_info :  permet de connaître l'état du processus.
persistence :  demande explicite pour que les processeurs ne
          soient libérés que lors d'une opération _delete.
```

4.5 L'interaction entre processus

La figure 4.3 est une illustration de la relation entre les entités du modèle de processus de Parx. Il existe trois niveaux différents d'interaction entre ces entités : la communication entre *threads* d'une même *tâche*, la communication entre *tâches* à l'intérieur d'une *Ptâche*, et la communication entre *Ptâches* s'exécutant simultanément dans la machine.

Les *threads* à l'intérieur des *tâches* d'une même *Ptâche* interagissent par le protocole de rendez-vous sur des canaux, tel que nous l'avons vu dans la section 3.5.2. Il peut y avoir deux formes de gestion des canaux : soit ils sont déclarés dans le programme et leurs extrémités (le processus émetteur et le processus récepteur) sont complètement déterminées avant exécution, soit ils sont des objets du système et leur utilisation requiert un appel système et une gestion dynamique de celui-ci à l'exécution d'un programme.

La première solution est plus cohérente avec le modèle conceptuel de Parx, dans le sens où toutes les composantes d'un programme parallèle sont liées ensemble, par une étape d'édition des liens, pour constituer une seule *Ptâche* qui utilise les facilités de communication offertes par le noyau. Dans la seconde solution, un programme ne peut être complètement lié car ces composantes interagissent à travers le système d'exploitation. Toute action du système implique un surcoût d'utilisation des ressources telles que la mémoire et la puissance de calcul, la première solution est alors aussi la plus efficace.

Comme les *Ptâches* sont des programmes liés séparément, lorsque deux ou plusieurs *Ptâches* ont besoin d'interagir ensemble, elles doivent faire appel au système. Dans ce cas, il faut un objet de communication contrôlé par le système. Le *port* est alors bien adapté. En règle générale, les *ports* sont associés aux *tâches* d'une *Ptâche* (comme c'est le cas des *ports* associés aux acteurs dans Chorus), les *threads* d'une *tâche* peuvent alors retirer des messages des *ports* dont la *tâche* est la propriétaire. Tout *thread*, indépendamment de sa localisation, peut envoyer des messages à un *port* dont il connaît "l'adresse", autrement dit, s'il détient le droit d'accès en écriture. Les *ports* permettent de contrôler l'accès aux objets gérés par le système en utilisant des *capacités*, qui définissent à la fois le moyen de désignation et les droits d'accès.

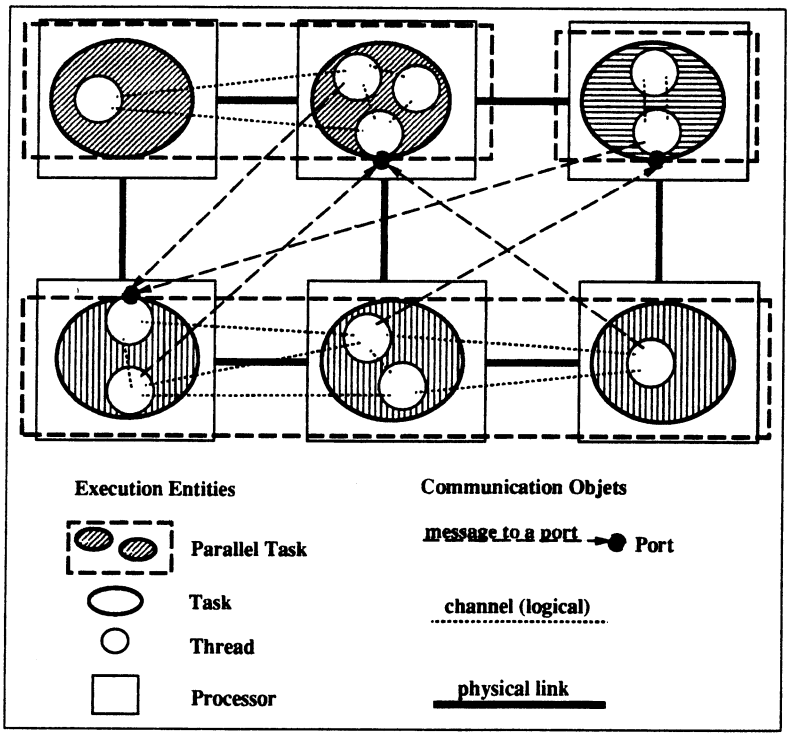


Figure 4.3: Relation entre entités et objets de communication de Parx.

4.6 La notion de *domaine de communication*

Nous associons la notion de *domaine de communication* à un *cluster* ; comme celui-ci est le support d'exécution d'une seule *Ptâche*, un *domaine de communication* est donc associé à une seule *Ptâche*. A l'intérieur d'un *domaine de communication*, le noyau assure la communication entre processus par des canaux. Il n'y a pas besoin du contrôle explicite des niveaux supérieurs du système d'exploitation. Cette absence de contrôle du système rend les communications intra-*domaine de communication* beaucoup plus efficaces.

La notion de *domaine de communication* a une autre conséquence, même s'il existe simultanément plusieurs *domaines de communication*, l'acheminement des messages et les protocoles utilisés sont définis et réalisés de façon indépendante pour chaque *domaine de communication*. Ceci réduit considérablement la complexité de l'acheminement des messages.

Les communications entre *domaines de communication*, c'est-à-dire entre *Ptâches*, passent nécessairement par un contrôle de plus haut niveau. Le contrôle de la communication devient alors nettemment plus complexe. A titre d'exemple,

considérons le cas où une *tâche*, s'exécutant dans une *Ptâche* ayant un *domaine de communication* DC_u, fait un appel système en envoyant un message à un serveur particulier ; la *tâche* connaît l'adresse d'un port d'accès mais pas la localisation physique ; le serveur aura été placé dans une machine virtuelle qui constitue un autre *domaine de communication* DC_s ; plusieurs problèmes se posent par cette "simple" action :

1. Comment retrouver la localisation du serveur ?

2. Existe-t-il un chemin permettant la communication entre les domaines DC_u et DC_s ; autrement dit, existe-t-il un lien, soit direct, soit indirect (un *lien virtuel* construit au moyen du routage), reliant les deux domaines ?

3. Si un tel chemin existe, y a-t-il d'autres domaines de communication à traverser ? Si oui, alors a-t-on le droit ?

4. Si un tel chemin n'existe pas, est-il possible d'en créer un ?

D'autres questions pourraient encore venir se rajouter ; le lecteur comprendra que les réponses à des telles questions impliquent des mécanismes de contrôle de plus haut niveau que ceux nécessaires à l'intérieur d'un seul *domaine de communication.* L'action inévitable du système sur les communications inter-*domaines de communication* fait que ce type de communication donne lieu à des protocoles plus lourds que ceux intra-*domaine de communication.* Dans cette thèse, nous ne sommes concernés que par ces derniers.

4.7 Architecture du noyau de communication

Parx suit la philosophie des autres systèmes en ce qui concerne les fonctionnalités du noyau ; celui-ci doit offrir les services strictement nécessaires au contrôle de l'exécution des programmes et de la communication entre processus. Sur chaque processeur doit exister donc un noyau minimal réalisant le modèle de processus et le modèle de communication.

Nous nous intéressons ici au noyau de communication, notamment à sa structure générale, qui fournit le cadre des études plus détaillées que nous présentons dans la troisième partie de cette thèse.

Le noyau de communication de Parx comporte deux niveaux. La figure 4.4 est une illustration. Le premier niveau est une virtualisation des moyens de communication entre processeurs ; il offre une interface homogène par laquelle la machine peut être vue comme un réseau de processeurs complètement connecté.

Pour ce faire, le noyau contrôle les liens de communication des processeurs ; le routage des messages et la reconfiguration sont les deux "armes" dont le noyau se sert pour offrir une machine virtuelle dans laquelle tous les processeurs peuvent communiquer entre eux, indépendamment de leur quantité et de la topologie du réseau qui les interconnecte.

L'acheminement de messages par routage requiert la définition d'une technique de commutation des données, d'une fonction de routage sans interblocages, et du contrôle de flux ; nous examinons tous ces aspects dans les chapitres 6 et 7.

Le second niveau assure l'interprétation des messages et la communication entre processus suivant un protocole déterminé. Nous lui consacrons le chapitre 8.

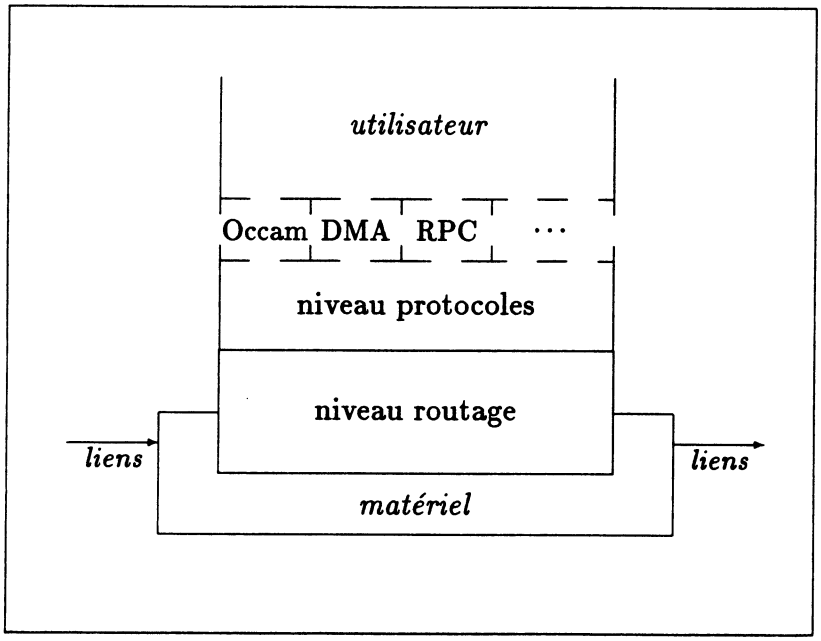


Figure 4.4: Noyau de communication de Parx, structure générale.

4.8 Conclusion

Le modèle de processus introduit par Parx , tout en s'appuyant sur les résultats de la recherche actuelle, apporte un concept nouveau absolumment nécessaire à la programmation parallèle : la *Ptâche.* Les notions de *tâche* et de *thread*, communément rencontrées dans d'autres systèmes, s'insèrent naturellement et de façon cohérente dans ce modèle.

Un soin particulier est accordé à la communication, par l'utilisation d'objets de communication différents on recherche l'efficacité. La définition de plusieurs protocoles permet de rendre un service complet et approprié aux différents besoins de communication des programmes à haut degré de parallélisme.

Le modèle proposé permet de construire un ensemble de services systèmes de base tels que le chargement de programmes, des serveurs de fichiers et gestion de la mémoire, entre autres. Ces services sont la base pour la mise en œuvre de politiques de gestion au niveau de sous-systèmes implantés en dessus de Parx.

Le modèle Parx offre une vue particulière de la machine, certains peuvent l'utiliser comme une machine Unix standard, d'autres pour développer des applications demandant de fortes puissances de calcul, d'autres encore pour évaluer des algorithmes parallèles.

L'ensemble des concepts et d'abstractions formulés est un tout cohérent ; il n'est pas une simple agglomération de toutes les tendances existantes. Un prototype de Parx a été développé dans le cadre du projet Esprit Supernode II, et l'implantation des sous-systèmes "Portable Common Tool Environment" (PCTE) et X/Open est actuellement en cours.

Chapitre 5

Autres systèmes : exemples et comparaisons

5.1 Introduction

Nous examinons ici la façon dont les concepts étudiés dans les chapitres précédents sont incorporés dans les systèmes d'exploitation. Nous considérons deux genres de systèmes : ceux orientés à l'exploitation des systèmes répartis et ceux orientés à l'exploitation des architectures parallèles sans mémoire commune. Même si les problèmes spécifiques diffèrent, les deux types de systèmes sont confrontés à l'exploitation du parallélisme.

Un *système réparti* est un réseau d'ordinateurs ne partageant pas de mémoire et communiquant au moyen d'un réseau local ou plus large. Chaque ordinateur est, en règle générale, un mini-ordinateur, un poste de travail ou une machine spécialisée. Un *système parallèle* est un ensemble d'unités appelées "nœuds" ne partageant pas de mémoire et communiquant par échange de messages. Chaque nœud est un ensemble autonome qui peut comporter un processeur central, de la mémoire privée et éventuellement des processeurs spécialisés pour le calcul flottant ou encore pour la communication.

Les systèmes répartis diffèrent des systèmes parallèles par plusieurs aspects : l'importance du couplage entre les sites ou entre les nœuds, l'ensemble de sites d'un système réparti est couramment hétérogène ; ils sont relativement indépendants, chacun pouvant démarrer, fonctionner et s'arrêter de façon autonome, leur réseau d'interconnexion est lent et pas toujours fiable, des sous-réseaux peuvent avoir des vitesses et des protocoles différents, il s'agit souvent des réseaux à diffusion. Par contre, les nœuds d'un système parallèle sont fortement couplés :

l'ensemble est plutôt homogène[1] ; ils démarrent, coopèrent et s'arrêtent généralement en forme coordonnée ; leur réseau d'interconnexion est normalement rapide, fiable et les connexions sont du type point à point. La qualification de réseau d'interconnexion "lent" et "rapide" est relative, elle fait ici référence au rapport entre la vitesse de traitement de l'unité central et la vitesse de communication des dispositifs physiques. Les systèmes répartis font généralement abstraction des caractéristiques topologiques du réseau ; celles-ci sont cruciales pour les machines parallèles.

Deux approches se distinguent quant aux systèmes d'exploitation pour les systèmes répartis [Krako87a][2] : dans la première, chaque machine du réseau a son propre système d'exploitation et une couche de communication permettant l'échange des données entre les machines est rajoutée par dessus ; la deuxième approche correspond à un système d'exploitation intégré dont une partie du système, couramment appelée "noyau", est dupliquée et s'exécute sur chaque machine. Selon Tanenbaum [TanRen85] c'est cette approche qui donne lieu à de vrais systèmes d'exploitation répartis.

Dans les sections suivantes nous examinerons trois systèmes d'exploitation répartis qui représentent l'état de l'art, et qui incorporent des concepts qui sont aussi à la base des systèmes parallèles. Ces systèmes sont : Amoeba, Chorus et Mach. Nous examinerons ensuite des systèmes parallèles, qui sont actuellement en développement, tout comme Parx, il s'agit des systèmes Peace et EDS (Pcl). Enfin, nous présentons aussi Helios, un des systèmes d'exploitation pour des machines à base de transputers.

Nous centrons notre étude sur les modèles de processus : les entités d'exécution offertes par le système, et sur les modèles d'interaction entre processus. Un bref aperçu des objectifs du système et de son architecture globale sera aussi inclu.

A la fin de ce chapitre, nous présentons des aspects comparatifs des systèmes parallèles. L'objectif est de montrer la position conceptuelle de Parx par rapport aux autres systèmes.

---

[1] Ce n'est pas obligatoire mais c'est le cas le plus courant.

[2] En réalité S. Krakowiak présente trois classes, nous avons réuni deux d'entre elles :*systèmes client-serveur* et *partage de ressources par interconnexion des systèmes homogènes* dans la première approche.

5.2 Amoeba

Amoeba est un projet mené conjointement par Vrije Universiteit (VU) et le Centrum voor Wiskunde en Informatica (CWI) à Amsterdam [Tanen89, RST89, Mullen89, Mullen90, RTM90]. Son objectif est de produire un système réparti intégré pour des machines hétérogènes dans un réseau local : stations de travail, processeurs de calcul, serveurs spécialisés et passerelles vers d'autres systèmes et réseaux plus larges. Les processeurs de calcul forment un pool de ressources utilisé au fur et à mesure des besoins. Les serveurs spécialisés comprennent un service de répertoires, de fichiers et de blocs disque. Différents réseaux locaux "Amoeba" peuvent être interconnectés, via les passerelles, pour former un système intégré plus large ([ReSta86], [RTSH87]).

5.2.1 Architecture système et modèle d'exécution

Chaque site, qui peut être un multi-processeurs, exécute un noyau minimal qui implante les appels systèmes et gère trois types d'objets : les *segments*, les *threads* et les *processus*. Un segment est une portion de la mémoire. Un *thread* est un flot d'exécution représenté par un compteur ordinal et un contexte limité à quelques registres d'état. Un *processus* est un espace d'adressage virtuel dans lequel peuvent s'exécuter un ou plusieurs *threads*.

Le modèle d'exécution est donc à deux niveaux. Le *processus*, qui s'assimile à notre définition de *tâche*, comporte toute l'information concernant l'administration d'un espace d'adressage divisé en segments, et de l'ensemble des *threads* s'exécutant dans cette espace d'adressage commun. Les *threads* d'un même processus partagent son espace d'adressage. La politique d'ordonnancement n'utilise ni les tranches de temps ni la préemption : un *thread* ne perd l'unité centrale que sur un appel bloquant au système. Sur un site multi-processeurs, plusieurs *threads* du même processus peuvent cependant être actifs en parallèle et doivent se synchroniser explicitement. Amoeba propose des sémaphores à cet effet. Il n'y a pas de relation de filiation entre processus ou entre *threads*, ce qui implique un modèle d'exécution plat.

5.2.2 Modèle d'interaction entre processus

Amoeba ne propose que des primitives de communication synchrones entre processus à travers un protocole basé sur des **transactions** [Mulder88]. Les objets de communication sont des *ports*. Une requête est émise d'un *port client* vers un *port de service* ; le serveur associé traite la requête et renvoie une réponse au port client.

Les *ports* d'Amoeba sont bi-directionnels, ils ont une partie émission et une partie réception ; les *threads* clients formulent des requêtes de service et attendent un message répondant à leur demande sur la partie réception de leur *port*. Chaque processus possède un ensemble de ports activés sur lesquels il peut recevoir des messages. La réception est bloquante.

L'utilisation de *threads* combinée à la notion de service permet de réaliser des schémas de collaboration entre serveurs. En particulier, un protocole de réplication de service a été développé. Il permet de regrouper des serveurs dans des classes fournissant des services équivalents. Un client peut alors s'adresser à un serveur quelconque de la classe, à un serveur particulier, ou à un sous-ensemble de serveurs.

5.3 Chorus

Chorus est un système d'exploitation réparti résultant des recherches menées à l'INRIA entre 1979 et 1986, poursuivies et étendues par Chorus systèmes. Le lecteur intéressé sur les détails de ce système peut lire par exemple [AGHR89], [Guill89], [ARSha89], ou encore [Chor89].

5.3.1 Architecture système et modèle d'exécution

L'architecture de Chorus a été conçue pour supporter différents types de systèmes d'exploitation ; elle est basée sur un noyau, qui réside sur chacun des sites du système réparti, et qui gère les ressources de base du site : mémoire et processeurs. Les sites peuvent être multi-processeurs.

Le noyau offre un modèle d'exécution à deux niveaux, basé sur des "acteurs" et des *threads*. Les acteurs supportent la machine virtuelle Chorus ; ils sont à la fois les unités de distribution et des espaces d'adressage protégés pouvant supporter chacun l'exécution de plusieurs *threads*. Les acteurs sont attachés à un site et à un seul et ne peuvent migrer, mais un site peut supporter plusieurs acteurs. Un *thread* est toujours lié à un seul acteur et un seul. Tous les *threads* d'un acteur partagent son espace d'adressage et lorsque le site est multi-processeurs, les *threads* peuvent s'exécuter en parallèle.

Par dessus le noyau Chorus on peut bâtir différents sous-systèmes. Chaque sous-système peut définir, en utilisant le support fourni par les acteurs, un ensemble de serveurs pour les ressources dont il fait usage ; il définit aussi un acteur gestionnaire de la sémantique de processus implantée par le sous-système et les appels

systèmes associés. Les acteurs créés à partir d'un sous-système lui appartiennent. Le code et les données des acteurs implantant des sous-systèmes sont placés dans l'espace d'adressage du noyau Chorus.

L'unité de gestion mémoire est le segment ; il peut être temporaire, sa durée de vie étant alors liée à celle d'un acteur ou d'un *thread*, ou bien persistant, sa durée dépassant celle des acteurs. L'espace d'adressage d'un segment est divisé en *régions* ; chacune faisant correspondre tout ou partie d'un segment à une adresse donnée de l'espace d'adressage d'un acteur. Différents acteurs peuvent définir des régions se recouvrant, auquel cas les segments sont partagés, même s'ils se trouvent sur des sites différents.

La politique d'ordonnancement ne distingue pas les *threads* d'un même acteur. La priorité de chaque *thread* est une combinaison de celle de son acteur et de la sienne propre au sein de ce dernier. L'ordonnancement est préemptif suivant des tranches de temps allouées à chaque *thread*. A chaque instant, c'est le *thread* de plus haute priorité qui s'exécute ; même les *threads* exécutant des appels système sont soumis à préemption. Les acteurs dont la priorité est supérieure à un certain seuil sont considérés *temps réel* et ne sont pas soumis à la règle des tranches de temps.

Chorus, aussi bien qu'Amoeba, offre un modèle d'exécution plat, sans relation de filiation entre ses entités.

5.3.2 Modèle d'interaction entre processus

Les *threads* à l'intérieur d'un acteur communiquent et se synchronisent en utilisant les méthodes classiques de la mémoire partagée. Pour l'interaction entre les acteurs, Chorus offre l'échange de messages à travers des *ports*. Les messages reçus par un *port* sont mis en file d'attente jusqu'à ce qu'ils soient consommés par un *thread* ayant le droit de recevoir des messages sur le *port*. Seuls les *threads* d'un acteur auquel un *port* est attaché peuvent recevoir un message sur ce *port*. Un *port* ne peut être attaché qu'à un seul acteur à la fois, mais il peut migrer au fur et à mesure qu'il est attaché à d'autres acteurs. Un acteur, par contre, peut avoir plusieurs *ports*. Les messages de la file d'attente d'un *port* peuvent le suivre dans ses migrations successives.

Les messages peuvent être échangés de façon asynchrone ou par invocation. Dans le premier cas, l'émetteur n'attend pas l'arrivée du message à destination, il n'a aucune garantie que le message sera effectivement délivré. Dans le second, l'émetteur est bloqué jusqu'à la réception d'une réponse du récepteur.

L'invocation est la base pour construire un protocole *client-serveur*.

Les *ports* peuvent être rassemblés en *groupes de ports*, offrant ainsi un service de diffusion. Une requête demandant un service peut être adressée à un groupe de *threads* fournissant un service équivalent. Aussi, un service particulier peut être sélectionné parmi un groupe de serveurs équivalents, la sélection se faisant par identification directe, ou par localisation sur un site supportant un objet donné.

5.4 Mach

Mach est supporté par le département de la Défense aux Etats-Unis, et a été choisi comme système d'exploitation par l'Open Software Foundation ; il a une longue histoire, qui remonte à Accent [RasRob81] et ses objectifs sont de supporter aussi bien les machines multi-processeurs que les systèmes répartis.

5.4.1 Architecture système et modèle d'exécution

De même que les systèmes déjà examinés, Mach est un système intégré, mais la différence est que le noyau de Mach fut conçu comme une sorte de noyau Unix ([Accet86], [TeRa87]) auquel il a été rajouté le support nécessaire à l'exécution de plusieurs *threads* au sein d'un même processus Unix et une gestion de mémoire virtuelle innovatrice, élément essentiel des abstractions du système.

Le modèle d'exécution est basé sur deux abstractions qui subdivisent un processus Unix en deux niveaux : la *tâche* et le *thread* [Teva87t]. Une *tâche* est l'environnement dans lequel les *threads* peuvent s'exécuter ; elle fournit un espace d'adressage virtuel et l'accès protégé aux ressources système. Une *tâche* comporte des objets de communication et comptabilise les ressources qu'elle utilise. Un *thread* représente l'exécution d'un flot d'instructions. Chaque *thread* s'exécute dans le contexte d'une et une seule tâche jusqu'à terminaison. Une tâche peut contenir plusieurs *threads*. Les *threads* d'une même tâche peuvent s'exécuter en parallélisme vrai sur un site multi-processeurs. L'ordonnancement des *threads* s'effectue par tranches avec différentes variantes, une complète présentation se trouve dans [Black90].

Mach offre un modèle d'exécution plat, la relation de filiation traditionnelle d'Unix étant remplacée par des droits d'accès sur une *tâche*.

Une troisième abstraction introduite dans Mach est l'*objet mémoire*, qui permet la gestion de la mémoire secondaire et notamment la gestion de mémoire

virtuelle. L'espace d'adressage et géré par *pages*. Une tâche peut allouer des *régions* ; celle-ci étant un intervalle d'adresses consécutives. Des *gestionnaires de pages* sont associés aux pages physiques. Ils reçoivent un message lors de chaque faute de page concernant l'une des pages dont ils ont la charge, ou lorsqu'il faut les écrire sur support de mémoire secondaire. Chaque page peut être protégée en lecture, écriture et exécution. De même que pour Chorus, la mémoire virtuelle est essentielle dans Mach [ARSha89] et [Rash87].

5.4.2 Modèle d'interaction entre processus

Les objets de communication sont appelés *ports*, mais ils sont définis comme étant de simples canaux de communication unidirectionnels implantés comme des files d'attente de messages gérées et protégées par le noyau. Cette définition est très restrictive par rapport aux *ports* d'Amoeba et davantage encore en ce qui concerne les *ports* de Chorus.

Les primitives de communication correspondent aux opérations d'émission (**send**) sur un *port*, auquel cas le message est rajouté à la queue ; et de réception (**receive**) sur un *port*, auquel cas le message en tête de la queue est délivré. Si aucun message n'est disponible, la primitive **receive** est bloquante. Les opérations d'émission et de réception peuvent être combinées dans un seul appel système.

Plusieurs *threads* peuvent avoir le droit d'émettre sur un *port* mais un seul a le droit de recevoir. Lors de l'émission d'un message, un *thread* peut, soit décider d'attendre que le message soit déposé dans la file d'attente du *port*, soit continuer son exécution et être prévenu a posteriori par un message généré par le noyau. Le premier cas correspond à une communication synchrone, le second à une communication asynchrone.

Les *ports* de Mach permettent la désignation de tâches indépendamment de leur localisation physique. Ils permettent aux *threads* d'échanger des messages quelle que soit leur localisation et sans qu'ils aient à se préoccuper de celle de leur correspondant. Des droits sont requis pour opérer sur des *ports* ; ils sont collectifs pour tous les *threads* d'une même tâche. Une seule tâche possède le droit de réception sur un *port*, mais plusieurs de ses *threads* peuvent demander à recevoir simultanément. Les droits d'accès aux *ports* peuvent être transmis dans un message. Un *port* peut être restreint ou non. Dans le premier cas, il doit explicitement être désigné par son nom ; dans le second, il fait partie de l'ensemble de *ports* par défaut sur lesquels sont attendus des messages.

5.5 PEACE

Le nom PEACE [WSP90] provient de *Process Execution And Communication Environment* ; tel que Parx et à la différence d'Amoeba, Chorus et Mach, il s'agit d'un système ayant comme objectif primaire le support du développement et d'exécution des programmes dans une architecture multi-processeurs sans mémoire commune. L'architecture cible de Peace est SUPRENUM [BGM86].

Conçu dès le départ avec cet objectif, Peace propose fondamentalement les services de communication par échange de messages et la gestion de processus à très bas niveau ; les services classiques tels que gestion des fichiers et gestion des ressources doivent être fournis par des systèmes existants s'exécutant sur une machine hôte ; l'idée de fond est que toute la puissance de l'architecture multi-processeur soit mise au profit des applications. Le principal souci est de concevoir un noyau de système de communication efficace.

5.5.1 Architecture système et modèle d'exécution

Trois concepts sont à la base de l'architecture de Peace : l'*entité* ("*entity*") est l'unité permettant la répartition des modules d'un programme ; le *processus* est l'unité d'exécution séquentielle à l'intérieur d'une *entité* ; le *message* est l'unité de communication entre processus.

Une *entité* encapsule un programme application dans une *machine abstraite* lui fournissant une interface d'exécution à travers un ensemble de primitives ; dans le cas de SUPRENUM les primitives sont directement inspirées des constructeurs du langage Fortran et des primitives de communication par échange de messages. La machine abstraite est composée de trois parties : une partie appelée *"service coupling"* s'occupant de la désignation des objets et fournissant une localisation transparente à la configuration du système, une deuxième partie contenant les librairies système et la troisième partie, qui gère le modèle d'exécution, appelée *"runtime system"*.

Le modèle de processus de Peace a trois niveaux ; les abstractions définies sont appelées *Leagues*, *Teams* et *Lightweight Processes* (ou *thread*). Un *Team* définit un environnement d'exécution commun à des *threads* projetés sur un même espace d'adressage. Les *Teams* permettent alors d'isoler et de répartir des groupes de *threads*. Un seul *Team* peut encapsuler plusieurs *entités*.

Un programme application dont les diverses parties sont réparties[3] est constitué par un groupe de *Teams*. Une *League* est un groupe de *Teams* ; elle fournit un environnement où les communications entre processus sont très efficaces. Mais les *Leagues* sont aussi un moyen de supporter plusieurs programmes application répartis, autrement dit, des multi-utilisateurs, car l'ensemble de *Teams* d'une *League* est isolé[4] de l'ensemble de *Teams* des autres *Leagues*.

La notion de *League* ajoute un niveau dans le modèle de processus par rapport aux systèmes que nous avons examiné jusqu'à présent. Une *League* est assimilable à la *Ptâche* de Parx.

5.5.2 Modèle d'interaction entre processus

Le paradigme de base est l'échange de messages entre processus. Peace offre un objet de communication appelé *port*. Deux processus communiquent directement à travers un *port* ; la communication est synchrone et sans stockage intermédiaire des messages. Les primitives de base sont **send** et **receive**.

Sur la base des communications synchrones, trois services (protocoles) sont offerts :

- *invocation*: correspond au modèle synchrone et bloquant :

  émettre_requête-attendre_réponse

  qui permet de réaliser des appels de procédure à distance et d'implanter un modèle *client-serveur*. Ce paradigme se trouve dans tous les systèmes déjà vus. Chaque processus possède un espace de mémorisation limité, dans lequel sont conservés les arguments et l'identificateur de l'appel de procédure. Le récepteur accepte explicitement de recevoir un message d'un émetteur quelconque, le traite et retourne une réponse, débloquant ainsi l'émetteur. Les messages sont reçus dans leur ordre d'arrivée sur le site récepteur. N'importe quel processus du *Team* peut répondre. Au retour, le résultat de la procédure remplace les paramètres d'appel chez l'émetteur.

- *transfert de gros volumes de données* : réalise le transfert de segments d'une taille quelconque. Le transfert est réalisé entre deux espaces d'adressage se

[3]Dans l'article que nous avons pris comme base pour cette section, on ne trouve jamais des mots ou des concepts tels que parallélisme ou concurrence, nous ne voulons pas rajouter ces concepts dans notre description, bien qu'il nous semble regrettable que la présentation d'un système d'exploitation pour une machine parallèle ait oublié complètement le parallélisme.

[4]Le mot isoler devrait signifier protéger car il s'agit clairement d'un problème de gestion mémoire.

trouvant n'importe où dans le réseau. Encore une fois il n'y a pas de stockage intermédiaire, le transfert n'est alors initié que si le récepteur a reservé l'espace mémoire nécessaire. L'émetteur désigne un processus destinataire, un couple d'adresses source et destination, et une taille. L'émetteur doit se synchroniser avec le récepteur avant que le transfert ait lieu, lors de sa terminaison l'émetteur est débloqué et il peut envoyer une réponse au récepteur.

- *émission synchrone* : il s'agit de l'émission synchrone de données de taille quelconque. Ce protocole est fourni pour des raisons de performances, car il élimine le besoin de synchronisation explicite du second. Il correspond à un protocole "allégé" permettant le transfert des messages entre processus du niveau utilisateur qui n'ont pas besoin du mécanisme client-serveur.

L'objet de communication est le *port*, sans capacité de stockage des messages et sans capacités associées. Par contre, dans Peace, un *port* a des *capacités de routage*. Le rôle d'un *port* de Parx est joué dans Peace par une **porte** qui est associée à chaque processus et qui lui permet de communiquer. Une **porte** est désignée par un nom global unique formé d'un identificateur du nœud et d'un identificateur local ; il s'agit donc d'une méthode de désignation classique utilisée aussi dans les systèmes répartis.

Comme une **porte** est toujours associé à un processus, la désignation d'une **porte** est celle du processus. Cependant, des **portes** peuvent être dynamiquement associées aux processus, ce qui permet notamment la reconfiguration dynamique lors de la migration d'un *Team*, le noyau redirige automatiquement les messages destinés aux *Teams* ayant migré.

5.6 EDS (Pcl)

EDS provient du nom du projet *European Declarative System* concernant le développement d'un ordinateur multi-processeurs sans mémoire commune destiné à supporter des applications telles que : Bases de Données Relationnelles Parallèles, Lisp parallèle, ou encore Prolog parallèle. Le système est présenté dans [IsBor90].

5.6.1 Architecture système et modèle d'exécution

EDS développe PCL (Process Control Language), qui supporte un modèle de processus ayant trois abstractions de base : *threads*, *teams* et *tâche*.

Une *tâche* fournit un espace d'adressage virtuel (un processeur virtuel) qui peut être réparti sur plusieurs processeurs différents ; les *threads* s'exécutant dans une même *tâche* partagent l'espace d'adressage commun, même lorsqu'ils s'exécutent dans des processeurs différents.

L'ensemble de *threads* d'une même *tâche*, qui sont confinés dans un même processeur, constitue un *team* ; celui-ci est alors un environnement où les *threads* sont concurrents et s'exécutent en pseudo-parallélisme. Etant donné que les *tâches* dépassent les limites d'un processeur physique, plusieurs *teams* différents peuvent co-exister dans un même processeur. La figure 5.1 montre très bien la signification des entités de ce modèle.

Pour supporter des multi-utilisateurs, on introduit le concept de *job* qui représente en fait un utilisateur. Le *job* est une entité administrative dans laquelle on comptabilise l'utilisation des ressources ; il offre au programme utilisateur un environnement protégé avec les ressources dont il a besoin pour s'exécuter.

5.6.2 Modèle d'interaction entre processus

Les *threads* à l'intérieur d'une *tâche* communiquent en utilisant la mémoire commune correspondant à l'espace d'adressage de la *tâche*. Les *threads* des *tâches* différentes communiquent par échange de messages, même s'ils se trouvent dans le même processeur.

Un mécanisme appelé *Virtually Shared Memory* (VSM) sera implanté ; ceci signifie que la répartition physique de la mémoire sera cachée au niveau programmation d'applications et même au niveau d'implantation du modèle d'exécution. Ce mécanisme est théoriquement très puissant, surtout parce qu'il éviterait la

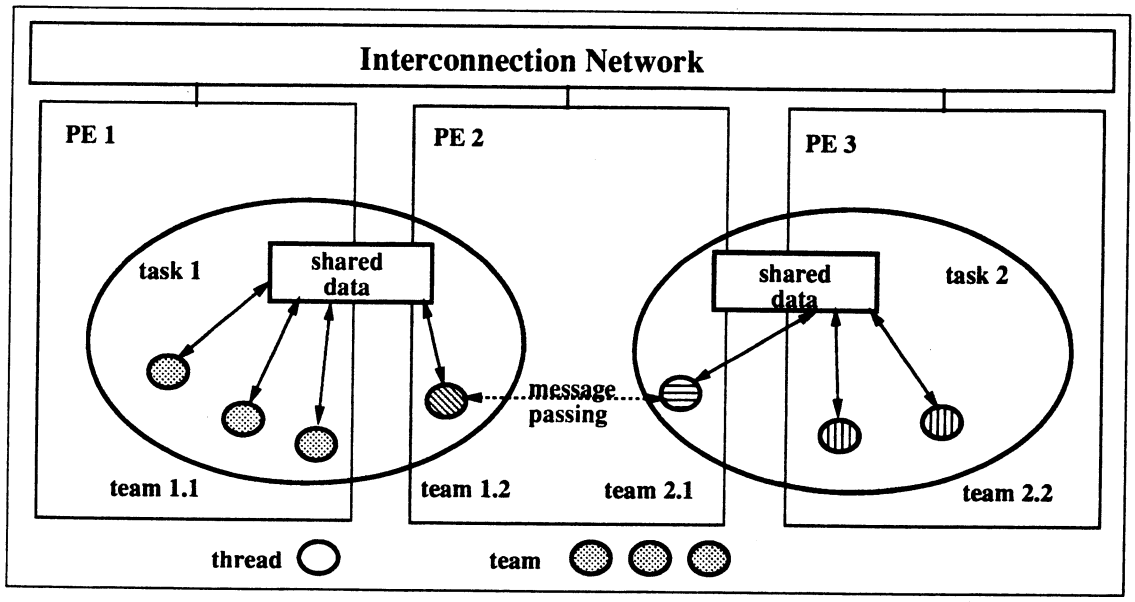


Figure 5.1: Architecture système EDS : Interaction entre les entités du modèle d'exécution.

copie, nécessaire lors de l'utilisation du modèle de communication par échange de messages, des grandes structures des données ; en effet VSM donnerait la possibilité d'accès selective à des parties d'une telle structure.

La VSM est à la base de la conception de Pcl. Reste à évaluer l'efficacite réelle car le modèle VSM n'est pas reflété par l'architecture sous-jacente, le transfert d'information par des moyens physiques de communication sera nécessaire même entre *threads* d'une même *tâche* appartenant à des *teams* différents.

5.7 Helios

Helios [Garnett] est un système d'exploitation, développé par Perihelion pour des réseaux de transputers. Helios fournit un compilateur C pour transputers et un environnement utilisateur à la Unix, qui a été porté sur plusieurs ordinateurs faisant office de machine hôte pour le réseau de transputers. Helios est devenu populaire parmi les programmeurs de ce type de machine ; il est en fait l'une des rares alternatives au système de développement offert par Inmos : le TDS, qui offre un bon environnement pour la programmation des applications parallèles, mais qui est très limité pour le développement du logiciel de base.

5.7.1 Architecture système et modèle d'exécution

A la différence des systèmes présentés ci-dessus, Helios traite explicitement des applications s'exécutant sur plus d'un processeur. Les processeurs sont considérés comme des ressources qui peuvent être initialisées à l'aide d'un noyau minimal appelé *nucleus*, qui contrôle l'allocation du processeur et de la mémoire, et met en œuvre un mécanisme de communication par messages.

Helios fournit d'autre part les outils nécessaires au chargement de programmes, une bibliothèque de fonctions système et un gestionnaire de processus mettant en œuvre l'abstraction de *tâche*.

L'architecture du système suit de très près le support système fourni par le transputer. Une *tâche* peut contenir plusieurs flots d'exécution, appelés *processus*, entièrement pris en charge par le matériel. Notons qu'un processus Helios est alors un processus léger et correspond à notre définition de *thread*. Le *nucleus* est composé de processus fournissant les services systèmes aux applications. Un même processeur peut supporter simultanément plusieurs *tâches* ; les processus des différentes *tâches* ne sont pas différenciés par le nucleus, ainsi donc, les *tâches* s'exécutant sur le même processseur ne sont pas protégées les unes des autres.

Au niveau utilisateur, un interpréteur de commandes reprend les fondements de celui d'Unix en rajoutant un support pour le parallélisme explicite. En créant des regroupements de *tâches* appelées "*task force*", l'utilisateur peut ainsi accéder à la programmation parallèle.

Helios est un système bâti sur le paradigme client-serveur. Le système comporte un ensemble de *tâches* "serveurs", typiquement un serveur de fichiers, accessibles par des objets de communication.

5.7.2 Modèle d'interaction entre processus

Les objets de communication sont appelés *ports* ; mais ils réalisent un protocole point à point synchrone entre deux processus, ce qui correspond vraiment à un canal ayant le même protocole des canaux occam. Ceci correspond essentiellement à ce que le transputer met directement en œuvre par le matériel, même sur les liens physiques.

Le système ne garantit pas l'acheminement correct des messages, et fournit un mécanisme de délai de garde pour se prémunir contre les pertes. Ceci nous semble être le plus grand handicap du système Helios.

5.8 Conclusion : analyse comparative des systèmes

Nous présentons ici une analyse comparative des modèles de processus et d'interaction entre processus proposés par les systèmes parallèles. Nous examinons particulièrement leur rapport avec les modèles proposés par Parx.

Les systèmes répartis présentés dans ce chapitre ont des aspects communs avec Parx car ils résultent d'une démarche similaire, mais plus orientée à des systèmes informatiques répartis qu'à des ordinateurs massivement parallèles. Ils ont en commun avec Parx un noyau minimal (idée qui provient d'ailleurs de ces systèmes répartis), reléguant au niveau de services systèmes les politiques d'administration des ressources. Le noyau se contente de gérer des processus et d'offrir les mécanismes de communication appropriés. Les services système sont fournis par différents sous-systèmes.

Une différence notable est dans le modèle d'interaction entre processus : les primitives de communication et de synchronisation. Le partage de mémoire est très utilisé dans Chorus et Mach, alors que Amoeba, comme Parx, préconisent une distribution plus effective. Une seconde différence est dans le support des programmes parallèles. Chorus et Mach ne sont pas conçus à la base pour une coopération étroite entre sites ou processeurs. Les mécanismes de base ne le prévoyant pas, un support de cette nature est peu à peu rajouté.

L'approche adoptée par Parx est le parallélisme massif, dès les fondements de l'architecture du système. Nous allons donc examiner, avec plus de détail, le rapport conceptuel entre Parx et les autres systèmes parallèles présentés dans ce chapitre.

5.8.1 Modèle de processus

Tous les systèmes parallèles éprouvent le besoin d'enrichir le modèle de processus classique à un seul niveau d'une ou de plusieurs entités. Plusieurs fonctionnalités sont couramment regroupées dans le processus traditionnel. Les différents modèles de processus des systèmes que nous venons d'examiner dans ce chapitre correspondent, en règle générale, à des combinaisons différentes de ces fonctionnalités.

Pcl

Le modèle de Parx comporte le même nombre de niveaux que celui de Pcl, avec des fonctionnalités semblables. Cependant, Parx nous semble plus simple et plus clair : Pcl offre des supports redondants, par exemple pourquoi placer des *threads* sur des processeurs différents si l'on souhaite qu'ils partagent la même mémoire virtuelle ? Est-il raisonnable de supporter des applications suffisamment monolithiques pour demander un partage de mémoire entre plusieurs processeurs ?

Parx choisit d'offrir la possibilité de partage physique de mémoire au sein d'une même tâche, et propose que la manipulation de structures de données très larges soit effectuée par un support spécifique à un sous-système, à l'aide des mécanismes de copie mémoire offerts par le noyau. Ainsi, le noyau n'a pas à se préoccuper de problèmes de cohérence.

Peace

A nouveau, nous retrouvons dans Peace un modèle de processus à trois niveaux. Cependant, cette fois-ci, la mémoire n'est pas partagée, et chaque niveau a un correspondant dans Parx. La différence fondamentale est que l'on accéde au code système par des appels de procédure à distance, alors que dans Parx, il est exécuté par des serveurs auxquels les utilisateurs envoient des messages.

Helios

Le modèle de processus Helios est à deux niveaux : processus et *tâches*. La "task force" n'a pas vraiment d'existence propre, elle peut être considérée comme un ensemble de commandes utilisateur pour créer un réseau de *tâches* communicantes. Ce niveau étant offert par l'interface de programmation par un langage de description : le "Common Description Langage". Le modèle de processus Helios est incontestablement moins riche et incomplet pour la gestion du parallélisme. Celui de Parx est plus cohérent.

5.8.2 Interaction entre processus

Bien que les systèmes reconnaissent ne pas pouvoir offrir tous les protocoles de communication et synchronisation dont auraient besoin les applications, il essaient de fournir un ensemble de protocoles de base. Deux protocoles se retrouvent régulièrement : le client-serveur et le tranfert de gros volumes de données.

Pcl

Pcl propose le *port*, qui est une file d'attente de messages, de taille quelconque, qui peuvent être lus par un *team*. Chaque *port* a un nom global, plusieurs extrémités d'émission, une de réception. L'émetteur fournit avec son message un identificateur permettant au serveur de lui répondre. Le noyau ne garantit pas la délivrance des messages, mais l'émetteur est prévenu lorsqu'elle ne peut se faire. Pcl propose en fait deux protocoles. Le premier est un protocole de partage des pages mémoire. Les pages peuvent être lues et écrites indépendamment de leur copie originale, un protocole de validation permettant ensuite de synchroniser les différentes copies. Le second est le protocole par ports, avec ou sans réponse.

Ces deux protocoles sont supportés dans Parx. La sémantique de partage de pages mémoire de Pcl est lâche : l'utilisateur a la responsabilité d'en assurer la cohérence. Le protocole par *ports* est flexible, mais ne garantit pas l'acheminement correct des messages. D'autre part, l'utilisateur a trop de visibilité sur la gestion des tampons. Parx fait le choix de conserver l'information dans la mémoire de l'émetteur, afin de ne pas avoir à effectuer une gestion de tampons en réception comme Pcl.

Peace

Il propose trois protocoles de communication, qui sont en fait des variations du même protocole de base : l'invocation. Peace a rajouté des protocoles de communication différents du modèle client-serveur (typique des systèmes répartis). Celui-ci tout en étant très bien adapté aux besoins des systèmes ne répond pas efficacement, et parfois conceptuellement, aux besoins des programmes application. Les protocoles proposés par Peace sont couverts par Parx à travers deux protocoles différents : accès direct mémoire et client-serveur.

Peace propose aussi la migration de *teams*, en autorisant des *ports* à servir de relais d'indirection vers une nouvelle localisation. Parx ne propose pas de migration automatique par le noyau, parce que les informations de relocation ne sont pas toutes sous sa responsabilité. Il se contente de fournir un ensemble de mécanismes permettant la migration.

Helios

Helios appelle *port* ce que dans Parx est un canal : ils réalisent un protocole de rendez-vous point à point synchrone entre deux processus. La taille maximale d'un message est un paramètre de dimensionnement des tampons du logiciel de routage. Des "surrogate ports" permettent de relayer des messages de

processeur en processeur jusqu'à leur destination finale. Helios ne garantit pas l'acheminement des messages, et fournit un mécanisme de délai de garde pour se prémunir contre les pertes.

Il présente deux lacunes importantes. L'espace mémoire occupé par le logiciel de communication est directement proportionnel à la taille maximale de message autorisée. D'autre part, l'acheminement de messages n'est pas garanti, et l'utilisateur doit utiliser un délai de garde. Cependant, il ne peut borner raisonnablement ce délai, qui dépend de multiples facteurs évoluant dynamiquement : charge du réseau, vitesse relative des correspondants, etc.

L'approche adoptée par Parx nous semble plus robuste, complète et évolutive. Le noyau de communication, tout en offrant les mêmes services que celui de Helios, permet de bâtir la plupart des protocoles requis par diverses applications.

Partie III

Contrôle de la communication dans les machines parallèles

Cette partie comporte trois chapitres.

Le chapitre 6 est une étude sur le routage des messages dans les architectures multi-processeurs sans mémoire commune. Nous examinons un ensemble de sujets liés au contrôle de la communication par routage des messages : les techniques de commutation des données, la structure d'un noyau de communication, la définition d'une fonction de routage. Dans tous les sujets nous analysons l'état de l'art et nous proposons des solutions appropriées à l'architecture cible. A la fin du chapitre, nous examinons les possibilités d'utilisation conjointe de la reconfiguration et du routage des messages.

Le chapitre 7 présente une étude des problèmes d'interblocage dans le routage des messages. Nous examinons les différentes méthodes, nous les comparons et nous proposons un nouvel algorithme qui est bien adapté aux architectures massivement parallèles.

Le dernier chapitre de cette partie est consacré à l'étude des protocoles de communication entre processus. Nous proposons une structure générique d'interprétation des protocoles par le noyau système et nous examinons en détail la conception du protocole synchrone, point à point, couramment appelé rendez-vous, *étendu à un réseau quelconque de processeurs.*

Chapitre 6

Routage des messages

6.1 Introduction

Dans le chapitre 4 nous avons montré la structure générale du noyau de communication. Elle comporte essentiellement deux niveaux : le routage des messages et l'interprétation des protocoles. Dans ce chapitre, nous abordons les problèmes qui concernent le premier niveau. L'objectif est de virtualiser les mécanismes de communication offerts par le matériel, c'est-à-dire d'offrir un *réseau de communication virtuel* qui émule un réseau complètement connecté. Notre hypothèse de travail est qu'un tel réseau virtuel doit permettre le transfert correct des messages entre tous les processeurs d'un *domaine de communication* à réseau d'interconnexion statique présentant une topologie quelconque.

6.1.1 Cahier des charges

Le cahier des charges que nous proposons ci-dessous comporte, en accord à notre hypothèse de travail, les fonctionnalités de la couche de routage des messages.

1. *Aucun message ne doit être perdu ou dupliqué* : il faut éviter que les procédures de gestion des ressources associées au routage (notamment la gestion des files d'attente) ne prennent, d'elles-mêmes, la décision d'éliminer un message ou d'en dupliquer un.

   Les protocoles construits au-dessus du routage pourraient contrôler aussi bien la perte de messages que leur duplication. Pour ce faire, ils peuvent utiliser des délais de garde –accompagnés de la réémission des messages– et l'étiquetage adéquat des messages. Cependant, le coût impliqué dans ce type de contrôle peut être important, aussi bien en temps d'exécution qu'en utilisation de ressources, d'où l'intérêt d'éviter que la couche de routage puisse perdre ou dupliquer des messages.

2. *La stratégie de routage doit être sans interblocage et sans famine* : les routes suivies par les messages ne sont pas nécessairement disjointes, par conséquent, les liens de communication et les tampons de stockage sont des ressources partagées. Les procédures associées au routage doivent se concevoir de manière à éviter l'interblocage entre les messages qui partagent ces deux ressources. Tous les messages doivent recevoir un service équitable : toute demande d'émission doit être traitée au bout d'un temps fini.

3. *Tout message doit être acheminé vers sa destination en un délai de temps fini* : il ne suffit pas d'éviter l'interblocage et la famine, il faut encore que :

   - la fonction de routage soit complète, c'est-à-dire qu'elle fournisse une route entre toute paire de processeurs du réseau.
   - les routes soient construites, autant que possible, sur les plus courts chemins. Ceci parce que le délai de transfert d'un message, ou latence, dépend directement de la longueur de la route suivie entre le processeur source et le processeur destination.
   - des congestions localisées sur certaines parties surchargées du réseau soient évitées par un contrôle de flux adéquat. Ces congestions ont pour effet d'augmenter la latence et de diminuer le débit global du réseau.

4. *La stratégie de routage doit être indépendante de la topologie et de la taille du réseau* : ceci est important notamment lorsque le réseau peut avoir une connectique quelconque, non régulière. Cette indépendance doit se manifester spécialement sur la quantité de ressources nécessaires au routage sans interblocage. Nous reviendrons sur ce sujet dans la section 7.2.

6.1.2 Les principaux problèmes de conception

Nous examinons ci-dessous l'ensemble des aspects impliqués dans le routage des messages.

1. *Techniques de commutation des données* : un premier aspect correspond à la méthode de *commutation des données*. Diverses techniques sont proposées dans la littérature ; elles montrent une évolution vers une utilisation plus efficace des liens de communication. Nous examinerons l'état de l'art dans la section 6.2.

2. *Fonctions de routage sans interblocage* : un second aspect est celui relatif à la *fonction de routage*. Indépendamment de la technique utilisée pour la commutation des données, les routes sont définies par la fonction de

routage ; elle est aussi responsable des interblocages. Nous allons proposer dans ce chapitre une fonction de routage qui est appropriée à la définition d'un algorithme de calcul des routes sans interblocage.

3. *Gestion des files d'attente* : la gestion des tampons de stockage est un problème qui se présente notamment lorsque la commutation des données est réalisée par stockage-et-réexpédition. Cette gestion a une influence sur le délai de transfert et sur le débit des messages ; elle peut se réaliser de plusieurs manières ; particulièrement, si chaque noeud du réseau est connecté à plusieurs voisins, il faudra gérer l'arrivée et l'émission de plusieurs messages en même temps. On parle de *gestion de files d'attente* et, comme nous le verrons dans la section 6.3, c'est la base du modèle du noyau de communication par routage.

4. *Le contrôle de flux* : le contrôle de flux est nécessaire pour éviter des problèmes de congestion pouvant conduire à des délais de transmission importants. La congestion se produit lorsque le taux d'arrivée des messages excède la capacité de service du réseau. Le contrôle doit s'exercer à plusieurs niveaux. Nous reviendrons sur ce sujet dans la section 6.6.

5. *La famine* : Dans le cahier des charges nous avons mentionné qu'il faut éviter la famine. Il s'agit là d'un problème plutôt lié à la mise en œuvre qu'à la conception générale des mécanismes de routage. Nous y reviendrons donc dans le chapitre 9.

Le reste de ce chapitre est organisé de la façon suivante : nous examinerons d'abord les techniques de commutation des données, ensuite, nous proposerons une structure du noyau de communication, suivie d'une étude des fonctions de routage qui nous conduit à en proposer une, appropriée à la résolution du problème de l'interblocage. Les aspects du contrôle de flux sont traités dans la section 6.6 et dans la section 6.7 nous examinons les possibilités d'utiliser la reconfiguration du Supernode de concert avec le routage des messages. L'interblocage est un problème étroitement lié au routage, il est cependant si important, que nous lui consacrerons un chapitre complet.

6.2 Techniques de commutation des données

Le routage des messages n'a pas ses origines dans les machines parallèles qui nous intéressent ici, mais il apparaît déjà vers les années 70 dans les premiers réseaux d'ordinateurs tels que : **TYMNET**, **ARPANET**, **TRANSPAC** [Sun77, QuiWa77, Quill80, SS80] ou encore l'**IBM SNA** [Ahu79a, Ahu79b]. Depuis cette époque, les techniques de commutation des données utilisées pour le routage des messages

ont été adaptées aux besoins des machines parallèles. Le but de cette section est de montrer leur évolution et leur état actuel.

6.2.1 Evolution des techniques

Originellement, les réseaux d'ordinateurs utilisaient comme moyen de communication des réseaux téléphoniques déjà existants. Le routage des messages fut donc réalisé sur la base d'un mécanisme utilisé en téléphonie, nommé couramment *Commutation de Circuit* (CC). Cette technique s'avérait cependant peu efficace, du point de vue du taux d'utilisation des liaisons physiques de communication, car basé sur la réservation de tous les liens du chemin entre la source et la destination du message ; une fois que le chemin était établi, les liens restaient réservés et très probablement inutilisés pendant de longues périodes de silence entre les correspondants. Le temps d'établissement d'un chemin était une autre cause d'inefficacité : lors de l'envoi fréquent des messages de faible taille, comme c'est souvent le cas dans l'échange d'information entre ordinateurs, ce temps devient trop important par rapport au temps de transfert.

D'autres mécanismes permettant de remédier aux déficiences de la commutation de circuit, furent proposés et sont largement utilisés aujourd'hui dans les réseaux d'ordinateurs ; il s'agit de la *commutation de messages* (CM) et de la *commutation par paquets* (CP), qui peuvent être groupées sous le nom de *stocker-et-réexpédier* (de l'anglais *store-and-forward*). Cette technique consiste à acheminer les messages, pas à pas, d'un processeur à un autre, tout le long du chemin entre le processeur source et le processeur destination. Lorsqu'un message arrive à un processeur intermédiaire, il est entièrement stocké avant d'être réémis. Une fois qu'un message est stocké, le lien par lequel il est arrivé est libéré et peut être utilisé par un autre message : les liens sont alors utilisés plus efficacement. Toutes les routes peuvent être prédéfinies, par conséquent, le temps d'établissement lors de l'émission d'un message est nul.

La technique de stockage-et-réexpédition a été adoptée en principe pour les machines multi-processeurs ; elle présente néanmoins des inconvénients qu'il faut souligner :

1. Un message doit être entièrement stocké dans un nœud intermédiaire. Même si le lien de réémission est libre ; il restera inutilisé pendant toute la durée de la réception du message alors que la réémission pourrait commencer dès la réception de l'en-tête, qui contienne l'information nécessaire à la détermination du lien de réémission. La réémission du message est inutilement retardée.

2. Un ensemble de processus, chargé de la gestion du routage des messages, doit être exécuté sur chaque nœud du réseau. Lorsqu'un nœud ne comporte pas un processeur spécialisé en communication, c'est le processeur de calcul qui doit prendre en charge le routage des messages au détriment du temps consacré au calcul proprement dit. Ceci est un des handicaps des architectures à base de transputers, rappelons que pour la machine Victor, examinée dans la section 2.5, IBM a conclu qu'un processeur passe, en moyenne, un 37 % de son temps à réaliser des activités de routage des messages.

Les techniques CC, CM et CP sont présentées, analysées et comparées par Kermani et Kleinrock [Klein76, KeKle79, KeKle80].

6.2.2 Etat de l'art

Pour éliminer le premier des inconvénients énoncés ci-dessus, de nouvelles techniques ont été proposées : il s'agit du *Virtual Cut-Through*[KeKle79] et du *Wormhole* [DaSe86]. Ces deux techniques, de conception assez similaire, proposent d'aiguiller un message directement du lien d'entrée au lien de sortie, si ce dernier est libre, dès l'arrivée de l'en-tête. De cette manière, un message peut "traverser" un nœud intermédiaire sans y être stocké, d'où une diminution de la latence.

La charge du réseau permet de réguler, d'elle-même, le comportement de ces techniques de commutation. Lorsque la charge du réseau est faible, la probabilité de trouver libre le lien de sortie est grande : le comportement se rapproche de celui de la commutation de circuit . Par contre, si la charge du réseau est forte, la probabilité de trouver libre le lien de sortie devient faible et le comportement se rapproche de celui du stockage-et-réexpédition.

C'est dans ce dernier cas de figure que les deux techniques agissent de façon différente. Lorsque dans un processeur intermédiaire un lien de sortie n'est pas disponible, *Virtual Cut-Through* propose de stocker tout le message en attendant que le lien soit libéré, la ressource la plus importante est donc la mémoire. *Wormhole*, pour sa part, bloque les liens qui sont encore occupés par le message en amont de la route ; c'est cette façon d'agir qui donne le nom à la technique, le message se comporte comme "*un ver de terre qui creuse une galerie qui se ferme après le passage du ver*", le message peut ainsi s'étendre sur plusieurs liens.

L'établissement d'une connexion directe entre un lien d'entrée et un lien de sortie n'est pas toujours possible dans un processeur, même dans le cas des processeurs "conçus pour les architectures parallèles", tel que le transputer. Les deux techniques presentées ci-dessus ne sont applicables qu'à des réseaux dont

les nœuds contiennent un processeur spécialisé en communication ; c'est la base pour éliminer le second inconvénient.

> *"La gestion du routage des messages dans un nœud d'une machine à architecture multi-processeurs sans mémoire commune doit être confié à un processeur spécialisé ; c'est la seule manière de ne pas pénaliser la puissance de calcul du nœud".*

Le Torus Routing Chip (TRC)

La technique *Wormhole* fut proposée pour un processeur spécialisé en communication qui réalise aussi une fonction de routage sans interblocage, c'est le *Torus Routing Chip* [DaSe86] ; il s'agit d'un circuit VLSI autonome, conçu pour la gestion du routage des messages dans des réseaux d'interconnexion en forme de tore.

La fonction de routage sans interblocage réalisée par le TRC découle directement de la topologie ciblée : elle se base sur l'algorithme *e-cube* que nous examinerons dans la section 7.5.

Un TRC permet de router un message suivant deux dimensions ; plusieurs TRC peuvent cependant être connectés en cascade pour permettre le routage sur une grille à n dimensions. La fonction de routage consiste donc tout simplement à respecter l'ordre des dimensions. Le TRC peut aussi bien être utilisé dans les réseaux en hypercube, il a été d'ailleurs conçu pour l'architecture *Cosmic Cube*.

La nouvelle famille des transputers : le T9000

Les transputers actuels incorporent des facilités élémentaires de communication qui sont largement dépassées par les besoins réels des architectures multi-processeurs ; ils sont en réalité le cas typique des deux inconvénients énoncés à la page 118 : les liens de communication permettent de réaliser des canaux de communication entre deux processeurs voisins mais le multiplexage des canaux logiques et le routage des messages sont à la charge du processeur de calcul.

D'autres limitations, comme par exemple le manque de gestion de la mémoire, ont été souvent soulignées par les constructeurs de systèmes pour des machines à base de transputers, ces limitations sont étudiées en détail dans [Langue91], nous allons donc limiter nos commentaires exclusivement aux aspects liés à la communication.

La nouvelle famille de transputers annoncée pour l'année 91, dont le composant principal est appelé transputer T9000, éliminerait les défauts actuels. Dans le

T9000, chaque canal physique est multiplexé par le matériel entre des canaux virtuels (ou logiques) ; chaque canal virtuel est représenté par une structure des données dans la mémoire du T9000, on peut alors déclarer un nombre de canaux qui n'est limité que par la mémoire disponible. Deux transputers T9000 peuvent être connectés directement par des liens physiques et le multiplexage réalisé par le matériel permet d'avoir une quantité virtuellement infinie de canaux entre eux.

Cette technique permet le partage des canaux physiques entre deux processeurs voisins, mais ne résoud pas le problème de routage des messages à travers le réseau ; un composant séparé joue le rôle de processeur spécialisé en communication : le C104. C'est lui qui permet de libérer le processeur T9000 de la gestion de routage des messages, la communication n'altère plus la puissance de calcul. Les T9000 et les C104 peuvent être interconnectés suivant une topologie quelconque. A l'aide du C104, les canaux virtuels sont réalisables sur tout le réseau.

Le C104 utilise la technique *Wormhole* et une fonction de routage que nous discuterons dans la section 6.4 ; différente de celle du TRC. Le C104 est un circuit VLSI capable d'interconnecter jusqu'à 32 paires de liens par un réseau crossbar : un seul C104 suffit pour réaliser des réseaux à 8 transputers T9000. La mise en cascade des C104 permet de réaliser des réseaux de taille quelconque ; selon les spécifications actuelles d'Inmos, la traversée de plusieurs C104 n'altère pas significativement le débit de bout en bout, le délai de transfert des messages sur un hypercube à 64K processeurs serait seulement le double de celui sur un hypercube à 64 processeurs.

Comme nous l'avons souligné dans la section 3.5.2, le protocole de communication par échange synchrone de messages, ayant seulement deux correspondants, ne suffit pas aux besoins réels des applications parallèles ; ceci est corroboré par les constructeurs du T9000 qui ont décidé d'incorporer dans le matériel un protocole *m vers un* dans lequel peuvent participer plusieurs émetteurs et un récepteur répartis sur le réseau ; le protocole du type client-serveur peut ainsi se réaliser plus efficacement.

6.3 Structure de la couche de routage

Les techniques *Virtual cut-through* et *Wormhole* ne peuvent être utilisées sur les transputers actuels ; la raison essentielle est qu'il est impossible d'interconnecter directement un lien d'entrée à un lien de sortie. La réception d'un message sur un lien implique la copie du message en mémoire ; de la même façon, l'envoi d'un message sur un lien implique la lecture du message directement de la mémoire. Le routage par stockage-et-réexpédition est cependant bien adapté et la première

couche du noyau de communication est structurée en conséquence.

Gestion des files d'attente

Si un processeur a plusieurs liens d'entrée et plusieurs liens de sortie, on doit décider de la technique à utiliser pour gérer les files d'attente face aux liens de sortie. Quatre techniques ont été proposées et analysées par Irland dans [Irland78] ; elles sont aussi reprises par Gerla et Kleinrock dans [GeKle80].

Soit : N_l le nombre de liens de sortie, T_f la taille maximale d'une file et M la capacité totale de l'espace mémoire destiné au routage. Les techniques sont les suivantes :

1. *files totalement indépendantes* : chaque lien de sortie a sa propre file d'attente. Les paramètres respectent la relation : $T_f = \frac{M}{N_l}$.

2. *une seule file partagée* : il n'existe qu'une seule file de longueur $T_f = M$, partagée par tous les liens.

3. *files partagées avec longueurs modifiables dynamiquement* : les valeurs de M et de T_f sont modifiées dynamiquement pour optimiser la longueur des files en fonction de la charge des liens. Dans ce cas, les paramètres respectent la relation : $\sum_{i=1}^{N_l} T_{f_i} \leq M$. La quantité $(M - \sum_{i=1}^{N_l} T_{f_i})$ est une portion de M partagée par tous les liens.

4. *files partagées avec longueur fixée par approximation heuristique* : il s'agit d'approcher le comportement optimal mais par une valeur fixe de T_f. Dans [Irland78] il est demontré que la meilleure approximation est réalisée par $T_f = \frac{M}{\sqrt{N_l}}$.

Dans [Irland78] on trouve notamment une comparaison des quatre techniques du point de vue de leur influence sur le délai de transfert et le débit de transmission des messages. Les courbes présentées par Irland montrent que le délai introduit par la technique de files totalement indépendantes est nettement inférieur à celui introduit par les autres techniques. Quant au débit des messages, cette technique est assimilable à celle des files partagées avec longueur modifiable dynamiquement, les deux ont un meilleur débit, par rapport aux autres techniques, lorsque le réseau est surchargé.

Outre les bonnes caractéristiques de la technique des files totalement indépendantes, elle est la plus appropriée pour être utilisée sur les transputers parce que :

1. elle permet de profiter au maximun, et d'une façon naturelle, du fait que les liens du transputer peuvent agir en parallèle, indépendamment les uns des autres et indépendamment aussi de l'unité centrale. Si chaque lien de sortie a sa propre file, il n'y aura pas de conflit d'accès et le parallélisme est complètement exploité.

2. elle permet d'éviter l'*interblocage direct* [Günt81], sans qu'il y ait besoin d'autres mécanismes.

Structure générale du niveau routage

A partir de la gestion de files totalement indépendantes nous sommes en mesure de définir une structure du noyau de communication pour le routage ; une illustration fonctionnelle est présentée dans la figure 6.1.

Un processus **récepteur** est associé à chaque lien physique d'entrée, il est chargé de recevoir les messages qui arrivent au processeur. Tout message reçu est aiguillé vers un processus **routeur** qui place le message soit dans l'une des files associées aux liens de sortie, soit dans la file locale associée aux protocoles ; la décision, qui dépend de la destination du message, est prise suivant une fonction de routage adéquate, que nous définirons dans la section 6.4. Un processus **émetteur** est associé à chaque lien de sortie, il prend un message de la file et l'émet sur le lien. Une fonction de type **récepteur** permet d'acheminer les messages locaux vers le **routeur**. Les messages locaux sont ceux envoyés par un protocole et qui entrent dans le réseau de routage.

Interface offerte par la couche de routage

La couche de routage, qui rappelons nous, réalise la fonction de communication de l'extension matérielle du noyau de Parx, offre une interface très simple. L'objet manipulé, c'est-a-dire l'unité d'information acheminée à travers le réseau, est le *paquet*, qui correspond couramment à un "message" de taille fixe et qui comporte un *en-tête* et un *corps*. L'*en-tête* contient l'identification du processeur destination du message.

Le routeur accepte les paquets en provenance de la couche des protocoles et les achemine suivant la même politique qu'il applique aux paquets reçus sur les liens. Les protocoles n'ont qu'à déposer les paquets dans une file d'où il seront pris par le routeur. Tout paquet étant arrivé au processeur destinataire est déposé par le routeur dans une file locale où les protocoles viennent prendre les paquets.

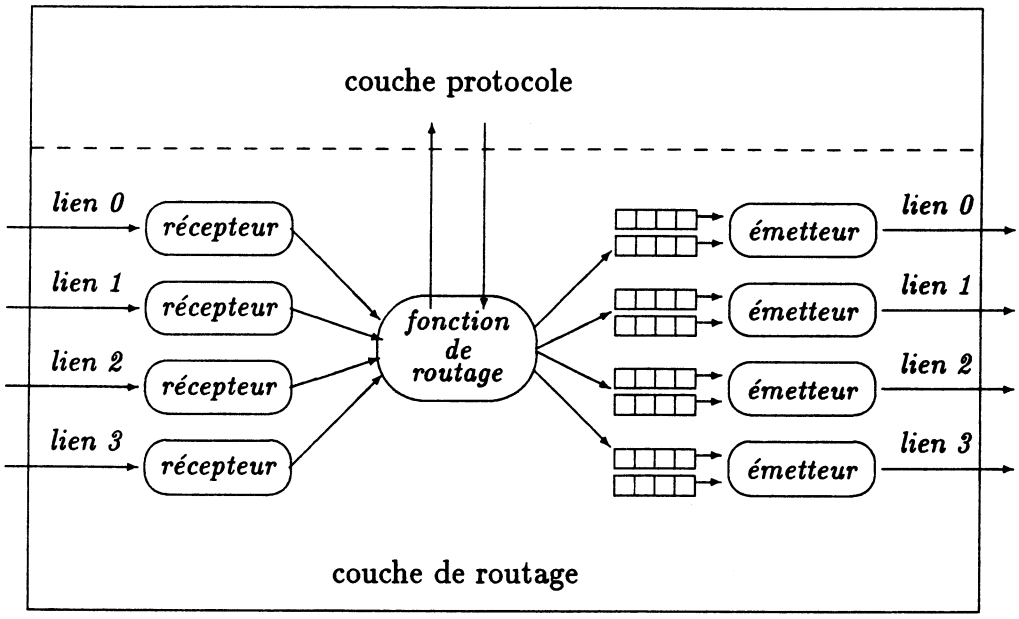


Figure 6.1: Structure du niveau routage

La description donnée ici est purement fonctionnelle, plusieurs solutions peuvent être adoptées lors de la mise en œuvre. Ces solutions sont discutées dans le chapitre 9. Ce qui nous intéresse d'examiner dans le reste de ce chapitre est la fonction interne du routeur.

6.4 Fonction de routage

La fonction de routage définit la procédure qui est utilisée pour acheminer un message, à travers le réseau, entre un processeur source et un processeur destination. Lorsque les processeurs source et destination ne sont pas voisins, le message doit traverser des processeurs dits intermédiaires ; dans ceux-ci, le routage consiste à aiguiller le message vers un lien de sortie appartenant au chemin qui conduit le message vers le processeur destination. La question essentielle est : *comment choisir le lien de sortie* ? C'est la façon dont on réalise ce choix qui conduit à des solutions différentes, que nous analyserons ci-dessous.

Définition de route

Nous considérons le problème de routage à l'intérieur d'un *domaine de communication*, lequel correspond à un *cluster* ayant une configuration physique

représentée par un graphe connexe, $RP = (P, L)$. Le routage des messages dans RP consiste à trouver au moins un chemin entre chaque paire de processeurs du réseau. Soit p_s le processeur source d'un message et p_d le processeur destinataire. Une route r de p_s à p_d est une suite de liens que le message doit traverser pour aller de p_s à p_d. $R(p_s, p_d)$ est l'ensemble de toutes les routes de p_s à p_d. Soit $r \in R(p_s, p_d)$, la longueur de r correspond à la cardinalité de l'ensemble de liens qui la composent : $\mathcal{L}(r) = \mid r \mid$. La distance entre p_s et p_d est : $d(p_s, p_d) = Min_{\forall r \in R(p_s, p_d)}(\mathcal{L}(r))$. Une route entre p_s et p_d avec $s \neq d$, est dénotée comme suit[1] :

$$r = route(p_s, p_d) = l_x^s, \ldots, l_y^k, \ldots, l_z^d$$

où :

$0 \leq x, y, z \leq N_l - 1$: N_l est le nombre de liens du processeur.

$0 \leq s, k, d \leq N - 1$: N est le nombre de processeurs, s est le processeur source, d est le processeur destination et k est un processeur intermédiaire.

Les diverses méthodes de routage

Les méthodes de routage peuvent se classifier en *déterministes* et *adaptatives* [2]. Le déterminisme implique que les routes soient calculées a priori et qu'elles ne soient pas affectées par les variations du trafic. L'adaptation, par contre, implique que la route suivie par un message est déterminée en fonction du trafic "réel". La maintenance, la propagation et la mise à jour de l'information concernant le routage, sont des activités absolument nécessaires au routage adaptatif [Stern79].

Lorsqu'on doit concevoir un noyau de communication, la question de la méthode à utiliser est inévitable. Alors, comment choisir ? La réponse a deux facettes : l'une relative aux mérites comparatifs de chaque méthode et l'autre relative à certains besoins particuliers imposés au routage.

Du point de vue des mérites, dans [CDB79] on trouve un excellent travail de simulation en vue de la comparaison des deux méthodes ; nous ne reprenons ici que les conclusions générales des auteurs :

- Le routage déterministe doit être utilisé sauf si l'une des conditions suivantes se présente :

[1] La notation utilisée ici a été définie à la page 31.

[2] Les qualifications de *statique* et *dynamique*, respectivement, sont préférées par certains auteurs [RuMu79], nous considérons qu'il s'agit de synonymes.

1. Le trafic est soumis à de fortes croissances soudaines.
2. Les besoins de communication qui déterminent le trafic sont inconnus.
3. Le réseau ne peut pas se configurer en fonction des besoins de communication.

- La variation du délai de transfert des messages est plus grande dans le routage adaptatif ; lorsque de longs messages sont découpés en paquets, qui sont acheminés indépendamment les uns des autres, même si le délai de transfert d'un paquet est plus faible par routage adaptatif, le délai de transfert de tout le message est normalement plus grand.

Si l'on tient compte des deux faits suivants : a) un réseau d'interconnexion reconfigurable, comme par exemple celui du Supernode, permet d'adapter la topologie de la machine aux besoins de communication des programmes parallèles et b) le découpage-réassemblage de longs messages fait partie des fonctionnalités typiques des noyaux de communication. Le routage déterministe devrait alors être utilisé.

Concernant le besoins imposés, nous allons considérer quelques-uns pour lesquels, même si les mérites de chaque méthode pourraient conduire à des indécisions, la méthode déterministe doit être nécessairement choisie. Ces besoins s'expriment ainsi :

1. le routage doit maintenir l'ordre d'émission des messages. Ceci simplifie la construction correcte des protocoles en dessus du routage, notamment lors de l'utilisation de la technique de découpage-réassemblage (voir la section 8.8).

2. le routage doit être prouvé correct, en particulier par rapport aux interblocages.

3. le routage doit être efficace en temps d'exécution et il ne doit pas gêner les programmes usagers.

La principale critique qu'on peut faire au routage purement déterministe est qu'il n'est pas capable de réagir vis-à-vis de situations de congestion ou de défaillance de certains éléments physiques du réseau. La solution se trouve dans la définition des routes alternatives, mais déterministes, qui permettent de résoudre ces deux problèmes ; elles permettent aussi d'équilibrer la charge des liens de communication pendant le fonctionnement en charge normale mais déséquilibrée. Le routage est alors essentiellement déterministe avec un degré limité d'adaptation.

C'est cette dernière solution que nous allons retenir. Aussi, dorénavant, lorsque nous parlerons de routage, il s'agira des méthodes déterministes.

Fonction de routage explicite et implicite

Les procédures de routage peuvent être divisées en *explicites* et *implicites*. Dans le routage explicite, la numérotation des processeurs et des liens de communication du réseau est arbitraire par rapport au routage, il faut cependant que l'information nécessaire à l'acheminement des messages, normalement sous la forme de tables, soit explicitement maintenue dans chaque processeur.

Dans le routage implicite, la numérotation des processeurs et des liens ne peut plus être arbitraire parce qu'elle est fixée par la fonction de routage elle-même. Ce type de routage se présente pratiquement sous la forme de *routage par intervalles* ([SaKha85], [LeTa87]). L'idée essentielle est d'étiqueter les processeurs et les liens du réseau, en les numérotant de telle manière qu'à chaque lien de chaque processeur on puisse associer un intervalle contenant l'ensemble de numéros de processeurs pouvant être atteints par ce lien. La fonction de routage consiste à trouver le lien de sortie dont l'intervalle contient le numéro du processeur destination du message.

Le routage par intervalles présente deux caractéristiques intéressantes, d'une part, il n'y a plus besoin de tables de routage, d'où un gain dans l'utilisation de l'espace mémoire consacré au routage ; d'autre part, il se réalise facilement dans le matériel : il n'a besoin que de quelques comparateurs dont la complexité est assimilable à celle d'un simple additionneur. L'inconvénient est cependant important : il n'est pas possible de trouver, pour toute topologie du réseau d'interconnexion, une solution avec un seul intervalle par lien, fournissant les routes sur les chemins les plus courts et sans interblocages. En règle générale, cette méthode s'applique bien à quelques topologies, notamment la grille et l'hypercube. Pour le tore, il faut déjà deux intervalles par lien pour maintenir la condition de non interblocage. A la limite, lorsque la méthode s'applique à des topologies quelconques, on peut arriver en extrême recours à avoir besoin de N intervalles, ce qui correspond à une table de routage utilisée par le routage explicite. Dans ce cas, les avantages ont disparus.

C'est cependant le routage par intervalles qui est incorporé dans les futurs transputers : le T9000 et le composant de communication C104 [Pount90]. Il y a deux raisons qui nous semblent justifier cette décision : d'abord la facilité de réalisation matérielle, ensuite le fait qu'on s'attend en principe à ce que tous les réseaux soient construits sur des topologies régulières et statiques acceptant bien la méthode de routage par intervalles. Ce dernier aspect se justifie, nous l'avons déjà discuté dans la section 1.3, par la faible latence introduite par le matériel lors de l'utilisation de la technique *Wormhole*.

Lorsque la fonction de routage est réalisée par logiciel et non pas dans le matériel, une méthode implicite introduit une latence plus grande que celle produite par l'utilisation de tables de routage. Le routage par intervalles requiert, comme son nom l'indique, la recherche d'un numéro dans un intervalle, et ceci consécutivement pour chaque lien de sortie jusqu'à trouver la solution[3]. Ce procédé est beaucoup plus coûteux en temps d'exécution que le seul accès mémoire nécessaire à la recherche d'une valeur dans une table.

Fonction de routage pour le noyau de Parx

En accord avec notre hypothèse de travail, nous devons utiliser une fonction de routage explicite. La fonction que nous proposons ci-dessous a été choisie en vue de faciliter la caractérisation et la résolution des problèmes d'interblocage que nous analyserons dans le chapitre 7. Nous allons définir formellement la fonction de routage par la relation R suivante:

$$\begin{array}{rcl} R : L \times P \times P & \longrightarrow & L \\ (l_i^s, p_s, p_d) & \longrightarrow & l_o^s \end{array}$$

Où l_o^s est le lien par lequel il faut réémetre un message arrivé au processeur p_s par le lien l_i^s ayant comme destination le processeur p_d.

Un message ne peut être réémis sur le lien par lequel il a été reçu, alors nécessairement $i \neq o$.

Une route ne peut contenir deux fois le même lien, sinon elle contiendrait un cycle et les messages suivant cette route n'arriveraient jamais à destination. Il faut remarquer que la restriction ne s'applique pas aux processeurs, seulement aux liens ; autrement dit, un message pourrait traverser plus d'une fois le même processeur en utilisant chaque fois des liens différents.

6.5 Sur l'espace mémoire nécessaire au routage

Dans le routage des messages par stockage-et-réexpédition il y a deux sources de consommation de l'espace mémoire : les tables de routage et les tampons de stockage nécessaires sur chaque processeur pour maintenir temporairement les messages en transit. On cherche à minimiser l'espace utilisé, dans le premier cas

[3]Aussi, le temps de recherche d'un lien de sortie étant variable, le délai introduit n'est pas constant, il dépend du lien de sortie et de l'ordre dans lequel on commence la recherche. Ceci devient plus important au fur et à mesure que le nombre de liens par processeur augmente.

au travers de la fonction de routage, dans le second cas au travers de la méthode de prévention des interblocages.

Taille des tables de routage

La fonction R requiert une table sur chaque processeur dont la taille[4] est $O(N)$. Elle permet néanmoins d'obtenir un algorithme de prévention des interblocages qui requiert une taille constante et réduite des tampons mémoire consacrés au stockage des messages en transit dans le réseau. Elle est aussi très efficace du point de vue du délai introduit lors du calcul du lien de sortie, un seul accès mémoire est suffisant.

Taille des tampons

L'espace mémoire requis par les tampons, M_t, dépend de deux paramètres : la quantité de tampons et la taille de chaque tampon. Le premier, qui correspond à la valeur M définie dans la section 6.3, est borné par : $N_l \leq M \leq f(N)$, où $f(N) \leq N$ est une valeur déterminée par le besoin de rendre le routage sans interblocage. Le second correspond à la taille du plus grand message qu'un tampon doit accepter ; normalement les messages qui circulent dans le réseau sont de longueur fixe : les *paquets*.

C'est donc la taille des paquets, l_p *octets*, qui détermine la taille d'un tampon, par conséquent, $M_t = M \times l_p$ *octets*. Dans la limite inférieure, lorsque la méthode de prévention de l'interblocage minimise le nombre de tampons, seulement un tampon par lien est nécessaire sur chaque processeur et $M_t = N_l \times l_p$. Comme N_l est normalement fixe, on ne pourra moduler l'espace mémoire utilisé que par la taille des paquets. Dans la limite supérieure, les méthodes de prévention d'interblocage demandent une quantité de tampons de l'ordre du diamètre du réseau.

Taille des paquets

La taille des paquets joue donc un rôle essentiel du point de vue de l'utilisation de la mémoire ; elle devrait être aussi petite que possible. La valeur l_p est cependant fixée par un autre besoin : la minimisation de la latence. La meilleure solution serait que tout message, indépendamment de sa taille, puisse être transporté dans un seul paquet. Les besoins sont donc contradictoires et le choix implique un compromis.

Une taille de paquet de 32 octets est proposée par Inmos dans la spécification du C104. Ce choix, induit sûrement par la grande quantité de mémoire que requiert

[4] $N \times N_l \times \log N_l$ (bits)

l'utilisation des canaux virtuels, s'éloigne fortement de ce qui est couramment utilisé, autour de 512 octets/paquet. Nous prenons cette valeur comme référence à utiliser dans Parx tant que les mesures de performance actuellement en cours ne déterminent pas une autre valeur.

L'importance de la taille des tables de routage

Supposons qu'on ait une machine ayant 128 transputers[5], la table de routage sur un processeur utilise 128 octets. Le cas le plus favorable pour les tampons correspond à celui où $M = N_l$, qui est possible exclusivement si M est indépendante de N. Prenons $N_l = 4$, si $l_p = 32$ *octets* alors $M_t = 128$ *octets*, si $l_p = 512$ *octets* alors $M_t = 2K octets$. Nous constatons donc que l'espace mémoire utilisé par les tables de routage ne devient important, par rapport à celui nécessaire aux tampons, que si la taille des paquets est faible.

Nous concluons que :

- Comme la taille des paquets ne peut être trop réduite, car elle a une influence sur le délai de transmission des longs messages, la quantité des tampons par processeur doit être indépendante de N et minimisée.

- L'espace mémoire utilisé par les tampons de stockage est nettement supérieur à celui nécessaire aux tables de routage. Celui-ci est en réalité négligeable par rapport à celui-là. En pratique, avec $l_p = 512$ *octets*, le rapport est de 1/16.

Comme nous le verrons dans le chapitre 7, la minimisation du nombre de tampons n'est pas gratuite, elle implique normalement d'autres coûts, comme par exemple l'utilisation de routes qui ne suivent pas le plus court chemin. Le problème est de trouver le meilleur compromis.

6.6 Contrôle de flux

Le contrôle de flux peut se réaliser à différents niveaux du noyau de communication [GeKle80]. Nous considérons les deux niveaux suivants :

- entre nœuds directement connectés.

- entre le nœud source et le nœud destination d'un message.

[5]De telles machines existent dans les laboratoires TIM3 et LGI à Grenoble.

Le premier, appelé *contrôle local*[PeSch75] est exercé au niveau du transfert des messages d'un processeur à un autre, il s'agit de limiter le taux d'émission d'un processeur afin de ne pas dépasser la capacité de stockage du récepteur. Ce type de contrôle est réalisé normalement au niveau du matériel par le protocole des liens de communication. Notamment dans le transputer, un nouvel octet ne peut être envoyé sur un lien tant que l'octet précédent n'a pas été acquitté. Un acquittement n'est bien sûr pas envoyé par le récepteur tant que l'octet n'a pas été dûment stocké.

Le second, appelé *contrôle de bout en bout*, est exercé par le processeur source au niveau des protocoles. Il a cependant une influence sur le comportement du routage, raison pour laquelle, nous le traitons ici en faisant abstraction des spécificités des protocoles. La méthode dont nous parlerons ci-dessous s'applique aussi au contrôle de la perte des messages, il s'agit des *accusés de réception* ou *acquittements*.

Un acquittement est un message spécial généré par un protocole lorsqu'il a reçu une information qui lui était destinée. Cette information peut être un message complet ou un paquet faisant partie d'un message ou encore un groupe de paquets. L'accusé de réception est bien sûr envoyé vers le processeur source de l'information.

Il y a deux méthodes pour renvoyer des acquittements ; soit on crée des paquets spéciaux envoyés indépendamment de tout autre paquet, soit on introduit l'acquittement sur un paquet déjà existant et ayant la même destination[6]. Cette dernière méthode n'est pas applicable toute seule : rien ne peut assurer qu'il y aura toujours un paquet ayant la même destination que l'acquittement.

Lors du contrôle de flux de bout en bout, la méthode des accusés de réception paquet par paquet est la plus stricte. Lorsqu'un message long a été découpé en plusieurs paquets, un paquet ne peut être envoyé que si l'on a reçu l'acquittement du paquet précédent. L'inconvénient est que le délai de transfert du message complet est fortement augmenté. Un mécanisme de *fenêtres*[7] permet d'envoyer plusieurs paquets sans attendre un accusé de réception pour chacun ; lorsque le taille de la fenêtre est réglable selon la charge du réseau, ce mécanisme permet de contrôler le flux tout en gardant des délais acceptables pour les longs messages.

La taille Ω de la fenêtre doit être calculée de manière à utiliser efficacement

[6] En anglais on appelle cette méthode "piggy-back" [Günt81].

[7] Outre les articles déjà cités dans cette section, ce mécanisme est étudié dans [Reis79] et [Ahu79a].

la capacité de stockage et la bande passante des liens de communication.

Pour que l'utilisation des fenêtres soit effective, il faut fixer une fenêtre par message, la taille doit être déterminée dans le processeur source et doit être communiquée au récepteur dans le premier paquet envoyé. Ω doit être précisément calculé empiriquement ; nous donnons cependant les bornes entre lesquels Ω peut varier :

$1 \leq \Omega \leq min(n_p, \mathcal{L}(r), T_f)$, où :

n_p : est le nombre de paquets dans lequel un message est découpé, une valeur $\Omega > n_p$ n'a pas de sens, une fenêtre qui puisse contenir tout le message serait suffisante.

$\mathcal{L}(r)$: est la longueur de la route entre le processeur source et le processeur destination. Cette limite n'est pas stricte, dans [Ahu79a] par exemple, il est montré que la taille de la fenêtre peut varier jusqu'à trois fois la longueur de la route.

T_f : est la longueur maximale des files d'attente, il n'est pas recommandable que Ω soit supérieure à T_f, car si l'on considère que tous les paquets d'une fenêtre peuvent être "instantanément" mis dans la file d'un lien de sortie, celle-ci sera complètement remplie et il restera encore des paquets dans la fenêtre qui n'ont pas de place dans la file. Le mot instantanément tient compte du fait que la réception des paquets en provenance de la couche des protocoles est beaucoup plus rapide que la réception ou l'envoi sur les liens physiques.

6.7 Routage des messages et reconfiguration dynamique

Dans une machine reconfigurable, le routage des messages est nécessaire lors d'une utilisation en mode reconfiguration statique[8]. L'utilisation du routage en mode reconfiguration dynamique n'est, par contre, pas évidente. C'est un sujet qui requiert une étude plus approfondie. Dans cette section, nous allons étudier quelques aspects relatifs aux possibilités d'utilisation concertée de la reconfiguration dynamique asynchrone et le routage des messages.

Considérons d'abord quelques définitions :

[8]Nous avons défini les modes de reconfiguration dans la section 2.6.

Configuration physique : Une Configuration Physique, dénotée CPh, correspond à une description précise de l'état des commutateurs reliant les processeurs ; elle est représentée par un graphe RP (voir la définition donnée à la page 31) dans lequel les sommets et les arcs sont dûment étiquetés pour identifier clairement les éléments physiques de la machine. Dans la figure 6.2

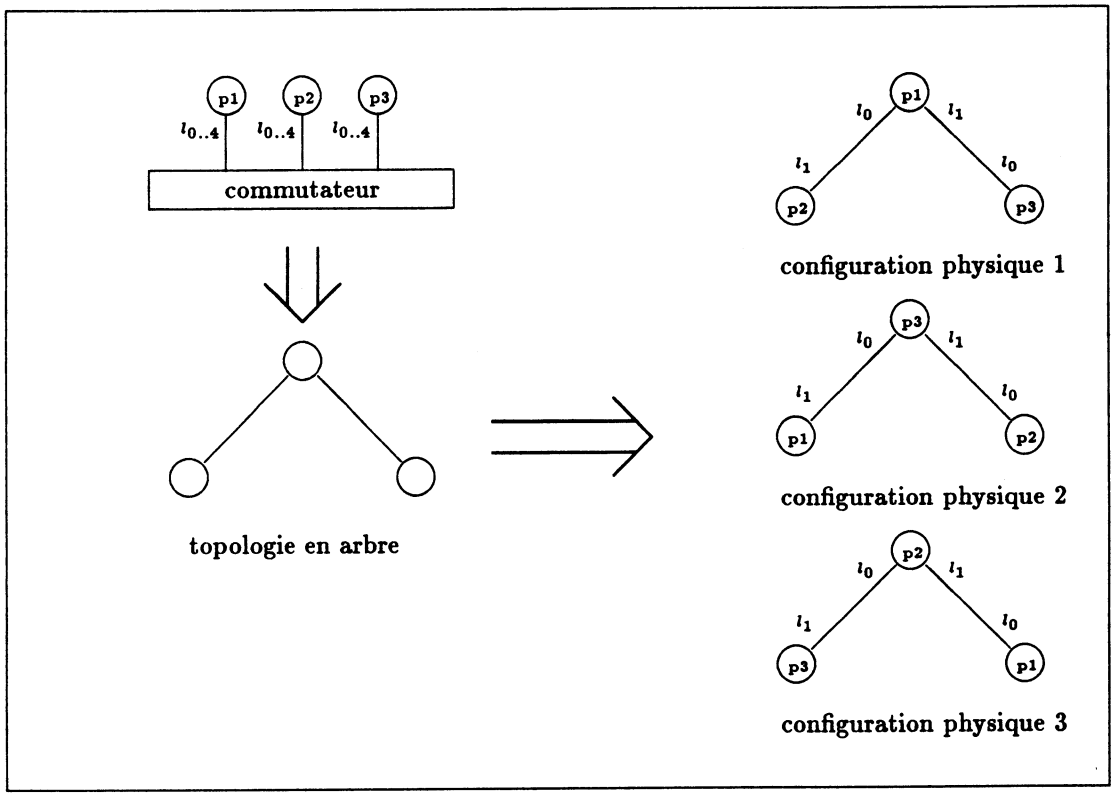


Figure 6.2: Illustration du concept de Configuration Physique

nous illustrons la signification du concept de configuration physique ; on peut constater qu'une même structure topologique peut donner lieu à des configurations physiques différentes.

Configuration :

Une configuration C est un état de la machine pour lequel une configuration physique et une fonction de routage sont définies : nous écrivons $C = (CPh, R)$. La fonction de routage, R, est celle définie à la page 128.

Reconfiguration :

Une reconfiguration RC est une fonction qui change une configuration en une autre :

$RC = C \longrightarrow C'$
où : $C = (CPh, R)$ et $C' = (CPh', R')$.

Lorsque $CPh = CPh'$ et $R \neq R'$, il s'agit d'une reconfiguration purement logique, qui consiste à changer les routes utilisées par le routage des messages sur une même configuration du réseau. Ce genre de reconfiguration présente un intérêt lorsqu'on veut essayer les performances des diverses formes des fonctions de routage en changeant l'attribution des chemins pour une même application.

Une reconfiguration physique implique $CPh \neq CPh'$. La fonction de routage originelle ne correspond plus au nouveau réseau d'interconnexion et l'absence d'interblocage pourrait ne plus être assurée. Une étape qui consiste à redéfinir R par le calcul des nouvelles routes s'avérerait alors nécessaire.

Dans la reconfiguration dynamique synchrone, le réseau d'interconnexion peut être considéré statique entre deux points de reconfiguration. C'est seulement dans le point de reconfiguration que l'on change la connectique de la machine et le changement est normalement significatif ; autrement dit, on est amené à changer fondamentalement la structure du réseau d'interconnexion sous-jacent. Lors d'une reconfiguration synchrone, toutes les connexions restent figées et supportent l'acheminement d'un nombre variable et virtuellement illimité de messages : on dit que les connexions sont *persistantes*.

Liens libres et liens persistants

La reconfiguration asynchrone est très différente de la reconfiguration synchrone : une demande de reconfiguration se produit exclusivement lorsqu'il y a un message à acheminer, dès que la connexion est établie et que le message est acheminé, la connexion est libérée, nous exprimons ceci comme une condition de *dynamicité* :

> *"Par reconfiguration asynchrone, une connexion n'existe que pendant l'acheminement d'un seul message et un seul, aucune connexion établie par ce type de reconfiguration n'est persistante*[9]*."*

[9]Rien n'empêche que le message soit composé de plusieurs messages "cumulés" dans un processeur et ayant la même destination ; si l'on pousse ceci à l'extrême, même des messages sur le réseau de routage pourraient être inclus : des routes peuvent être ainsi raccourcies ou des points de congestion peuvent être éliminés.

La configuration C d'une machine définit l'ensemble des connexions persistentes de celle-ci. Un lien physique appartenant à une connexion persistente sera appelé : *lien persistant.*

Un ensemble L_e est formé par tous les liens physiques, appartenant aux processeurs de la machine, qui ne sont pas utilisés dans la configuration C. Autrement dit, L_e est l'ensemble de tous les liens qui n'ont pas une connection figée. Ces liens ne sont pas considérés par la fonction de routage, par conséquent, ils ne sont pas utilisés à cet effet. Aucun des liens de l'ensemble L_e n'est associé à une connexion persistante, ils sont dits : *liens libres*. La problématique de la reconfiguration asynchrone est très différente selon qu'elle porte sur un lien persistant ou sur un lien libre. Nous y reviendrons à la page 140.

La gestion de la reconfiguration asynchrone doit comporter deux parties : une partie centrale localisée sur le transputer de contrôle du Supernode, et une partie répartie sur l'ensemble des processeurs du réseau. Nous parlerons de gestionnaire pour faire référence à une entité qui s'occupe de la reconfiguration. Il pourrait s'agir, par exemple, d'un serveur spécialisé faisant partie du système d'exploitation. Le gestionnaire central reçoit et traite les requêtes de reconfiguration qui lui sont envoyées par les gestionnaires locaux aux processeurs. Cette répartition est purement fonctionnelle et l'existence du gestionnaire central est une conséquence du fait que le commutateur du Supernode a été conçu avec commande centralisée.

Gestionnaire réparti : description des fonctionnalités

Sur chaque processeur du réseau il doit exister un gestionnaire de reconfiguration faisant partie du noyau du système. Ce gestionnaire ne peut décider d'une reconfiguration mais il peut seulement demander au gestionnaire central d'en réaliser une.

Le problème qui se pose à ce niveau, consiste à déterminer si une communication peut être effectuée plus efficacement par reconfiguration que par routage des messages. Lorsqu'un message doit être envoyé d'un processeur p_s vers un processeur p_d, il passe par le noyau de communication de p_s ; comme nous le verrons dans les chapitres suivants, l'*en-tête* du message contient normalement l'identification du processeur destination p_d. On peut dire donc que p_s "*sait*" qu'un message doit être envoyé vers p_d, il y a donc deux solutions: soit il envoie le message sur le réseau de routage, soit il demande au gestionnaire central l'établissement d'un lien direct entre p_s et p_d par la voie d'une reconfiguration. La question est : "comment prendre une décision permettant d'utiliser la solution pour laquelle

le temps global de communication est le plus court", autrement dit, celle qui rend la communication plus efficace. Il faudra alors évaluer le temps de chaque possibilité. Le problème est comment évaluer ces temps.

La fonction de décision

La prise d'une décision relève uniquement du processeur demandant la communication, le processeur p_s ; elle doit être aussi rapide que possible, autrement ce serait une source de retard de la communication ; prendre une décision rapide implique l'utilisation d'une fonction simple. La simplicité peut cependant entraîner une marge d'erreur trop importante. Nous allons analyser des modèles de mesure du temps pour la prise de décision.

Soit : $\mathcal{T}_{rout}$(s , d) une évaluation locale à p_s du temps que prendrait le transfert du message par routage, et $\mathcal{T}_{reconf}$ une valeur représentant le temps que prendrait le transfert du message par reconfiguration. On prendra la décision d'utiliser la reconfiguration au lieu du routage lorsque $\mathcal{T}_{rout}$(s , d) $> \mathcal{T}_{reconf}$.

L'évaluation de $\mathcal{T}_{rout}$(s , d)

Nous avons vu que le routage des messages est réalisé par la méthode de *stockage-et-réexpédition*, dans laquelle un message arrivant sur un processeur intermédiaire en chemin entre la source et la destination est entièrement stocké avant d'être réémis en aval du chemin. Le temps de transfert d'un message par routage peut être exprimé d'une manière générale par l'expression suivante :

$$\mathcal{T}_{rout}(\text{s , d}) = \mathcal{D}_{s,d} \times \kappa \times \mathcal{T}_i$$

$\mathcal{D}_{s,d}$ est la distance entre les processeurs p_s et p_d. Soit r la route déterminée par la fonction de routage, alors $\mathcal{D}_{s,d} = \mathcal{L}(r)$.

κ est une valeur représentant le mécanisme de *découpage-réassemblage* pouvant être utilisé pour envoyer des messages longs suivant une technique de commutation par paquets.

$\mathcal{T}_i$ est le temps de franchissement d'un processeur intermédiaire du chemin.

La signification de κ

Soit n_p le nombre de paquets dans lequel un message de longueur l_m est découpé, et soit l_p la longueur des paquets. Alors : $1 \leq \kappa \leq n_p$. Lorsque $l_m \leq l_p$, il n'y a pas de découpage et $\kappa = 1$.

Quand le message est découpé, la valeur de κ peut varier selon la technique de contrôle de flux utilisée (voir section 6.6). Supposons que la méthode de *fenêtres* est utilisée et soit Ω la taille de la fenêtre. Si l'on assume l'hypothèse que les paquets envoyés "à l'intérieur" d'une fenêtre constituent un seul message de longueur $\Omega \times l_p$, alors on a : $\kappa = \lceil \frac{n_p}{\Omega} \rceil$, où $\lceil \ldots \rceil$ dénote la valeur entière supérieure.

Les valeurs extrêmes sont :

$\kappa = 1$ pour $\Omega = n_p$ et $l_m = l_p \times n_p$, et

$\kappa = n_p$ pour $\Omega = 1$ et $l_m = l_p$.

Signification de $\mathcal{T}_i$

La valeur $\mathcal{T}_i$ ne peut être qu'une estimation car elle dépend de la charge réelle des liens composant le chemin de p_s à p_d. Il y a deux raisons principales pour que le processeur p_s n'ait aucune possibilité de connaître une valeur exacte : premièrement, p_s ne connaît pas la composition du chemin[10], seulement sa longueur ; deuxièmement, il ne peut avoir une "photographie instantanée" de l'état de tous les liens de la route mais seulement une "image" retardée.

Modèle théorique pour l'estimation de $\mathcal{T}_i$

Ce modèle suit celui de Kermani et Kleinrock [KeKle79, KeKle80]. La route entre le processeur source et le processeur destination est une suite de files d'attente, chaque file est représentative du mécanisme *stockage-et-réexpédition* implanté sur chaque processeur dans la route.

Le temps $\mathcal{T}_i$ est le même pour toute valeur de i si l'on considère les hypothèses suivantes : a) les messages arrivent sur le réseau indépendamment les uns des autres et avec une distribution Poissonienne, le taux d'arrivée est λ (messages/s), b) longueur des messages à distribution exponentielle, et c) utilisation équilibrée des liens de communication. Dans un processeur intermédiaire, lorsque le lien de sortie pour réémettre le message n'est pas libre, le message est mis dans la file d'attente. La valeur moyenne du délai, selon Kleinrock est :

$$\mathcal{T}_i = \mathcal{T} = \frac{\frac{\overline{l_m}}{B}}{1 - \lambda \frac{\overline{l_m}}{B}}$$

[10]Un processeur ne connaît que le lien au travers duquel il peut envoyer un message vers un processeur destination, il ne sait même pas qui est son voisin immédiat, autrement, chaque processeur devrait maintenir l'information de la configuration physique du *cluster* auquel il appartient et savoir se repérer lui-même et tout autre processeur dans la configuration.

où : $\overline{l_m}$ est la longueur moyenne des messages, et B est la bande passante des liens physiques en bits par seconde.

Estimation empirique de $\mathcal{T}_i$

Une méthode alternative consiste à calculer $\mathcal{T}_i$ par mesure de l'état du réseau. Nous n'envisageons pas d'avoir sur chaque processeur une mesure de l'utilisation de chaque lien du réseau, cela équivaut à avoir la connaissance de la configuration complète. Par contre, on peut avoir une valeur représentant la charge moyenne des liens du réseau par le temps moyen de service dans chaque processeur. Soit τ_i le temps moyen qu'un message passe dans un processeur entre le moment d'arrivée et le moment où il est réémis sur un lien ; la valeur de τ_i est évaluée par le processeur p_i par une méthode appropriée d'observation ("monitoring"). Cette valeur n'est pas affectée par la taille du message.

Soit τ_m la valeur du temps moyen de service qu'un processeur utilise pour calculer $\mathcal{T}_i$ au moment de la prise d'une décision. Cette valeur représente la valeur moyenne des temps de service dans le réseau et sa mise à jour suit, par exemple, la méthode utilisée pour la maintenance des horloges logiques ([Lamp78], [CaVi88]). Périodiquement, chaque processeur envoie à ses voisins immédiats la valeur τ_m. Un processeur p_i qui reçoit les valeurs de ces voisins met à jour sa propre valeur faisant la moyenne des toutes les valeurs qu'il a reçus en incluant la sienne, τ_i :

$$\tau_m = \frac{\sum_{voisins} \tau_m + \tau_i}{voisins + 1}$$

la nouvelle valeur est envoyée aux voisins[11]. Lorsqu'un processeur doit calculer la valeur $\mathcal{T}_{rout}$(s , d) il utilise la valeur de τ_m qu'il a actuellement pour calculer $\mathcal{T}_i$. La valeur du $\mathcal{T}_i$ est aussi affectée par la capacité des liens, ce qui s'additionne au temps de service ; $\mathcal{T}_i$ est donnée par :

$$\mathcal{T}_i = \mathcal{T} = \frac{\overline{l_m}}{B} + \tau_m$$

Evaluation de $\mathcal{T}_{reconf}$

Le gestionnaire local d'un processeur doit pouvoir estimer le temps de transfert d'un message lorsqu'il déciderait de le faire par reconfiguration. Il y a deux com-

[11] La maintenance d'une telle information se faisant par échange de messages, il est nécessaire de mesurer leur effet sur la charge du réseau. La génération et le recueil de l'information implique aussi l'exécution, sur chaque processeur, des processus additionnels ; leur effet doit aussi être évalué.

posants dans ce temps : avant que le lien soit prêt pour le transfert, c'est le temps de reconfiguration, et le temps d'émission proprement dit. Le temps $\mathcal{T}_{reconf}$ est donc : $\mathcal{T}_{reconf} = r + e$. Le temps r est affecté par trois facteurs : le temps de transfert des requêtes entre le gestionnaire local et le gestionnaire central, le temps de positionnement du commutateur lors d'une commande du transputer de contrôle et le temps moyen d'attente sur la file du gestionnaire central. Les deux premiers facteurs sont constants et déterminés par les caractéristiques physiques du bus de contrôle et du commutateur du Supernode ; ce temps sera dénoté par η.

Le gestionnaire central est séquentiel, car il est sur un seul transputer et il ne peut générer qu'une commande de commutation à la fois. Les requêtes, par contre, proviennent de tous les processeurs de la machine : le gestionnaire central de reconfiguration a affaire à tous les gestionnaires locaux. L'accès au gestionnaire central se fait donc au travers d'une file d'attente. Nous allons considérer que le temps moyen d'attente dans la file inclut le temps de service du gestionnaire. Ce temps que nous appellerons τ_r, ne peut être mesuré que par le transputer de contrôle lui-même.

Chaque gestionnaire local doit posséder une valeur τ_r actualisée lui permettant d'estimer le temps de service du gestionnaire central. Cette valeur permet la régulation de la production des demandes de reconfiguration : le taux d'arrivée des requêtes à la file du gestionnaire central doit être inférieur au temps de service, autrement la file croîtra indéfiniment ; ceci est automatiquement contrôlé par une valeur importante de τ_r ; dans ce cas les gestionnaires locaux auront tendance à ne pas émettre des requêtes.

Le temps r est donc : $r = \eta + \tau_r$. Le temps e dépend de la longueur l_m du message et de B : $e = \frac{l_m}{B}$. Une valeur moyenne de l_m ne peut être utilisée car la longueur des messages peut varier fortement. Le découpage en paquets n'étant pas justifié dans ce cas.

Les requêtes de reconfiguration

Le gestionnaire doit faire partie du niveau routage ; les protocoles lui passent les messages sans expliciter s'ils doivent être acheminés par routage ou par reconfiguration, le gestionnaire ne présente donc aucune interface spéciale. Il est néanmoins intéressant de donner aux utilisateurs la possibilité de demander une reconfiguration ; dans ce cas, l'interface est fournie par des requêtes qui ont la forme suivante[12] :

[12] la notation $\langle \ldots \rangle$ indique ici une option.

connecter $p_a\langle l_x, l_y, \ldots\rangle$ **à** $p_b\langle l_z\rangle$

Par cette requête on demande une connexion entre le processeur p_a et le processeur p_b, l_j représente un lien du processeur indiqué. Pour générer une telle requête, le programme utilisateur doit connaître la configuration de la machine sur laquelle il s'exécute.

La spécification des liens est optionnelle, normalement les liens ne seront pas indiqués pour laisser le choix au gestionnaire de reconfiguration, mais l'option est nécessaire lorsque des applications ont besoin de connecter certains liens d'une manière spécifique. Par exemple, un transputer qui a un lien connecté à un capteur externe pour la lecture des informations en temps réel ne pourrait permettre au gestionnaire d'utiliser ce lien pendant une reconfiguration. L'utilisateur, ayant la connaissance de la configuration de son *cluster*, peut proposer au gestionnaire l'utilisation de certains liens libres.

Le gestionnaire central

Nous avons déjà précisé que le gestionnaire central est essentiellement séquentiel et qu'il est placé dans le transputer de contrôle, car c'est lui seul qui commande le commutateur. Il prend les requêtes d'une file d'attente. Une requête doit être analysée et ses paramètres serviront à déterminer si elle peut être acceptée. Outre des erreurs telles qu'un processeur ou un lien inexistants[13] , une requête peut être repoussée car elle porte sur des liens qui ne peuvent être immédiatement interconnectés par le commutateur : rappelons que le commutateur à deux niveaux du Supernode est bloquant.

Un algorithme permettant de trouver le chemin dans le commutateur est coûteux ; cette phase du traitement des requêtes pourrait se paralléliser si un *cluster* associé au gestionnaire lui permet de lancer en parallèle (en vrai parallélisme) plusieurs instances de l'algorithme. Pour éviter des conflits entre les décisions, toutes les instances actives en même temps doivent collaborer. Le temps de service pourrait ainsi être réduit de façon significative. Ceci est un sujet à étudier dans le futur.

Reconfiguration d'un lien libre

Ce cas implique que le gestionnaire de reconfiguration a eu la possibilité de choisir des liens des processeurs p_a et p_b ne faisant pas partie de la configuration. Cette situation peut se présenter très souvent, considérons par exemple une topologie

[13]Un utilisateur peut soumetre une requête avec des paramètres erronés que le gestionnaire local ne saurait détecter, ou un processeur indiqué dans la requête n'existe plus car il a été éliminé pendant que la requête était dans la file d'attente.

en grille à deux dimensions, les transputers se trouvant dans la périphérie auront au moins un lien libre, et même deux pour les transputers se trouvant dans les quatre coins de la grille.

La reconfiguration portant sur des liens libres n'affecte pas le routage de messages. Le gestionnaire qui a demandé la reconfiguration peut commencer tout de suite l'émission du message, une fois qu'il a reçu un signal du gestionnaire central ; à la fin de l'émission, il envoie un message au gestionnaire central permettant de restituer la connectique de la machine. Durant cette opération, l'acheminement des messages par routage sur les liens persistants peut continuer à se réaliser correctement.

L'utilisation des liens libres par reconfiguration, implique que le noyau de communication de tout processeur actif doit accepter et traiter des messages arrivant par ces liens de la même façon qu'il accepte et traite les messages arrivant par les liens persistants.

Reconfiguration d'un lien persistant

Supposons que l'acheminement de messages soit basé principalement sur la reconfiguration asynchrone ; autrement dit, on essaie d'établir une connexion directe chaque fois qu'il faut acheminer un message entre deux processeurs. Ceci concerne donc les liens persistants et les liens libres.

Lorsque les demandes de communication deviennent trop fréquentes, la reconfiguration asynchrone peut devenir inefficace à cause du *temps de service de la reconfiguration*, comme il y a une partie du service qui est nécessairement centralisée à accès par file d'attente, si le taux d'arrivée des demandes dépasse constamment le temps de service, la file d'attente croîtra indéfiniment. Le temps de service est encore plus important lorsqu'il s'agit d'une machine Supernode à deux niveaux de commutation, car il est affecté par la complexité des algorithmes déterminant les points de connexion à utiliser dans le commutateur.

Le fait que la reconfiguration asynchrone soit plus efficace que le routage de messages à l'intérieur d'un *cluster*, dans lequel le placement de processus et la configuration ont été définis de façon à minimiser le coût de communication, est un sujet à étudier. Des évaluations pratiques sont nécessaires. Pour ce faire, et comme un apport aux nombreuses réflexions sur ce sujet, nous proposons des conditions permettant d'utiliser la reconfiguration asynchrone de concert avec le routage des messages.

notion de perturbation

> *"Lorsqu'une reconfiguration asynchrone porte sur une connexion persistante d'un* cluster, *nous dirons qu'il existe une* perturbation*".*

Une *perturbation* disparaît dès que la connexion persistante est rétablie à la fin du transfert du message.

Cette notion de *perturbation* n'est en réalité significative que pour le routage des messages. Un lien persistant est, selon notre définition, utilisé pour le routage intra-*cluster*, si la connexion d'un tel lien est modifiée par reconfiguration asynchrone, le routage des messages est *perturbé* ; plus précisément, le routage des messages par ce lien doit être complètement arrêté pendant la perturbation.

Les effets d'une perturbation

Lorsque la reconfiguration entraîne une perturbation, toutes les routes qui utilisent normalement les liens reconfigurés sont coupées. S'il y a plusieurs perturbations en même temps, le routage des messages peut devenir dangereusement perturbé. Les conditions suivantes s'imposent :

C1: *Aucun message ne doit être perdu à cause de la reconfiguration.*

C2: *Aucun message ne restera pour toujours dans le réseau sans atteindre sa destination.*

C3: *Le routage des messages est toujours sans interblocage.*

Pour assurer la condition C1 :

> *"Une condition* sine qua none *pour réaliser la reconfiguration physique des liens est qu'aucune communication ne soit active sur les liens impliqués."*

Lorsque le gestionnaire a acceptée une requête, il doit arrêter les communications en cours sur les liens impliqués. Supposons que la requête soit :

connecter $p_a\langle l_x\rangle$ **à** $p_b\langle l_y\rangle$

Les deux processeurs, p_a et p_b, ne sont pas les seuls touchés par cette situation. Pour montrer cela en détail, supposons que la situation avant reconfiguration était:

p_a , l_x **connecté à** p_c , l_w

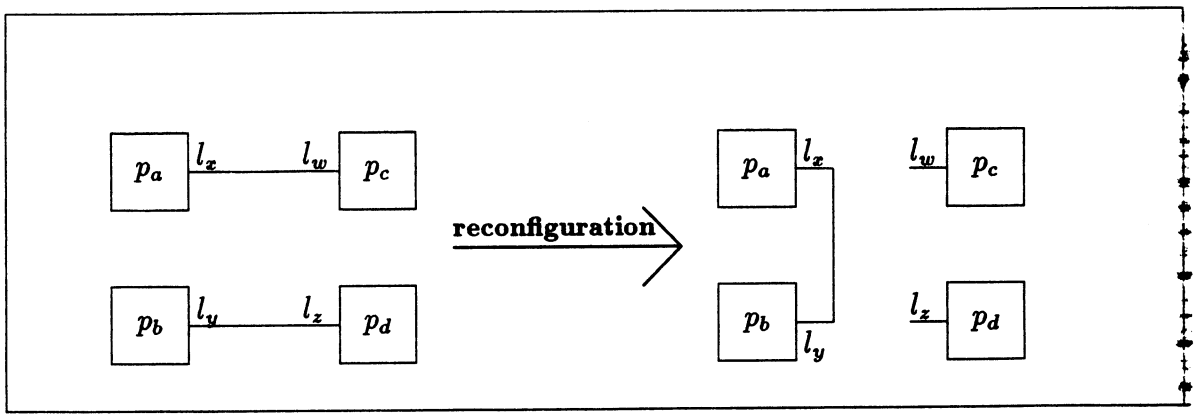


Figure 6.3: Exemple de modification de la connectique par reconfiguration asynchrone portant sur des liens persistants

p_b , l_y connecté à p_d , l_z

Lors de la reconfiguration, les liens l_w et l_z des processeurs p_c et p_d respectivement ne seront plus connectés, ils sont libres pendant la perturbation, les communications doivent nécessairement être arrêtées, aussi bien que celles en cours sur les liens l_x et l_y. Les quatre liens n'appartiennent plus à la configuration pendant toute la durée de la perturbation, ils ne pourront être réutilisés par le routage qu'une fois reconnectés. La figure 6.3 illustre ce que nous venons d'analyser. Si des demandes successives portent sur un même lien, les mêmes routes seront coupées successivement, la condition C2 requiert un contrôle sur une telle situation :

> *"Il faut éviter qu'un lien soit en reconfiguration de façon permanente, autrement certaines routes seraient coupées de façon permanente."*

A la différence de la reconfiguration synchrone, il est maintenant impossible de précalculer les routes à l'avance, d'autre part, le calcul des nouvelles routes pour modifier la configuration lors d'une perturbation n'est pas possible à cause du coût de cette action par rapport au temps d'émission d'un message. Pour accomplir la condition C2, un gestionnaire local doit tout simplement s'assurer qu'il n'y a pas un même message en attente d'être routé entre deux reconfigurations successives du même lien.

La condition C3 est remplie par le fait que la reconfiguration n'est pas persistante, la topologie du réseau d'interconnexion n'est jamais changée et la configuration C du *cluster* est par conséquent toujours valide.

Bilan

Dans cette section nous avons essayé de dégager, souligner et approfondir la problématique liée au contrôle de la reconfigurabilité du Supernode et sa relation avec le routage des messages. Nous avons franchi seulement un premier pas ; celui de proposer un ensemble minimun, cohérent et correct des fonctions pour la gestion des communication par reconfiguration dynamique.

Nos propositions doivent être considérées comme un point de départ. Des études plus approfondies, appuyées sur des mesures pratiques, sont nécessaires. Considérons un exemple : dans notre proposition, un message m, acheminé par routage, peut arriver à un processeur intermédiaire et se trouver temporairement bloqué parce que la route a été coupée par une reconfiguration. Si la perturbation est "trop longue", ce que l'on pourrait évaluer selon la taille du message envoyé par reconfiguration, on pourrait envisager de "rerouter" le message m en le "sortant" de sa route originelle, ceci signifie que m est envoyé sur un des processeurs voisins d'où il serait, probablement tout de suite, acheminé sur une nouvelle route. Ce qui semble très simple, ne l'est pas en réalité, l'action de "reroutage" met en péril les conditions C2 et C3, de nouveaux mécanismes devraient être considérés pour maintenir ces conditions.

6.8 Conclusion

Dans ce chapitre nous avons passé en revue les principaux aspects concernant le routage des messages dans les architectures multi-processeurs.

Une première conclusion concerne les techniques de commutation ; de nouvelles techniques ont été proposées afin d'améliorer les performances, c'est-à-dire diminuer le temps de transfert des messages. Ces techniques, qui ajustent leur comportement dynamiquement et automatiquement en fonction de la charge du réseau, sont nées des idées de Kleinrock en 1979 : le *Virtual Cut-through*. Quelque peu modifiée par Seitz, la technique est devenue celle du *Wormhole* qui est mise en pratique par le *Torus Routing Chip* et plus récemment par le circuit de routage C104 acompagnant le transputer T9000.

A la recherche d'une nouvelle diminution des temps de commutation des données, dans [JMY89], Jessophe, Miller et Yantchev proposent encore des améliorations au *Wormhole* par une technique appelée *mad postman routing*.

Outre l'amélioration des techniques de commutation, on a pris conscience du besoin de libérer les processeurs de calcul des tâches de routage et aussi de

l'inefficacité du routage par logiciel. On constate que les nouvelles générations de processeurs incorporent, de plus en plus, la gestion de la communication au niveau du matériel à l'aide des composants spécialisés en communication. L'effet de cette approche se fait sentir aussi bien sur les communications que sur l'utilisation réelle de la puissance de calcul des processeurs. Soulignons quelques conséquences :

- Le délai de transfert des messages par routage est fortement réduit par les nouvelles techniques et par l'utilisation des composants spécialisés. Ceci permettra d'augmenter la puissance réelle de calcul des architectures massivement parallèles.

- Il peut y avoir un changement dans la valorisation des différents paramètres qui caractérisent les réseaux d'interconnexion. Par exemple, le *diamètre* perdrait son importance vis-à-vis de la capacité d'expansion du réseau ; comme corollaire, l'hypercube perdrait une partie de son importance vis-à-vis de la simple grille à deux dimensions.

La structure du noyau de routage que nous avons proposé dans la section 6.3 suit les techniques habituelles et les fonctionnalités sont classiques, le point le plus important concerne la mise en œuvre que nous étudions dans le chapitre 9. Par contre, la fonction de routage, et notamment l'utilisation des ressources qu'elle implique, est encore un point de discussion ; nous avons examiné et critiqué les tendances actuelles dans la section 6.4, pour conclure sur l'utilisation d'une fonction relativement classique qui nous permet d'aborder plus aisément le problème de l'interblocage. Celui-ci est sans doute le plus important de tous, c'est la raison pour laquelle nous avons décidé de lui consacrer un chapitre complet, celui qui va suivre.

Chapitre 7

Interblocage dans le routage des messages

7.1 Introduction

Le problème d'interblocage apparaît à tous les niveaux de la structure de communication et sa résolution à un niveau donné ne préserve pas de leur réapparition à un autre. Nous distinguons trois niveaux : le routage des messages, les protocoles et le programme utilisateur. Le système doit éliminer la possibilité d'interblocage dans les deux premiers ; il lui est cependant impossible d'éliminer ceux introduits par une mauvaise programmation au niveau de l'utilisateur. Nous n'examinerons dans ce chapitre que l'interblocage au niveau routage.

L'interblocage est caractérisé par :

> *"l'existence d'un cycle de demandes*[1] *d'accès à une place dans un ensemble de files d'attente associées aux liens de communication. Si les files sont pleines, aucune demande ne peut être acceptée, aucun message ne peut avancer et aucune place ne peut être donc libérée".*

Plusieurs formes d'interblocage ont été répertoriées dans la littérature, en particulier, Günther [Günt81] présente différents types d'interblocage pouvant se produire par l'utilisation de la technique de stockage-et-réexpédition. Deux formes d'interblocage nous intéressent : l'*interblocage direct* et l'*interblocage indirect.*

[1] Un ensemble de demandes d'accès formant une relation cyclique est en réalité la caractéristique générale d'un possible interblocage dans toute circonstance où il existe des ressources partagées. La formalisation de cette relation cyclique prend cependant des formes très diverses, le lecteur désireux de connaître des formalisations différentes de celles données dans cette thèse, liées ou non au routage, peut lire par exemple : [Haber69], [ChaMi79], [Holt72], [GliSh80], [BroRo84], [Badal86] ou [RosDa86].

Le premier met en jeu seulement deux processeurs directement connectés par un lien de communication : supposons que deux processeurs, p_a et p_b, sont voisins et supposons que sur chaque processeur il existe une seule file partagée (voir 6.3). Si la file de chaque processeur est remplit de messages destinés à l'autre, aucun message ne pourra être envoyé par manque de place dédiée au stockage d'où l'interblocage direct ; il est éliminé par l'utilisation des files d'attente indépendantes, comme nous l'avons vu dans la section 6.3.

L'*interblocage indirect* met en jeu plus de deux processeurs, donc au moins trois liens de communication, son élimination, plus complexe que celle de l'interblocage direct, est le thème central de ce chapitre.

7.2 Cahier des charges des algorithmes

Dans ce chapitre nous examinerons plusieurs méthodes et algorithmes permettant d'éviter les interblocages dans le routage des messages, une base commune de comparaison est donc nécessaire. Dans la plupart des articles qui sont cités au cours de ce chapitre, les auteurs proposent les caractéristiques qu'un algorithme de routage doit présenter, elles sont normalement réduites à celles que les auteurs considèrent importantes selon leurs propres besoins. Nous avons cherché à regrouper le maximun de *propriétés souhaitables*.

Propriété 7.1 : Aucun message ne doit être perdu. *Nous avions déjà énoncé cette propriété dans le chapitre 6, section 6.1, pour le routage en général.*

Propriété 7.2 : Indépendance de la topologie du réseau. *L'algorithme doit être applicable à n'importe quelle topologie du réseau d'interconnexion.*

Propriété 7.3 : Indépendance de la taille du réseau. *La quantité de mémoire (tampons) consacrée au routage sans interblocage doit être constante et indépendante de la taille du réseau.*

L'importance des propriétés 7.2 et 7.3 énoncées ci-dessus se fait sentir particulièrement dans les machines massivement parallèles.

Propriété 7.4 : Utilisation efficace des tampons. *Lorsqu'un message demande l'accès à un tampon et qu'il y a des tampons disponibles, il faut éviter de refuser la demande.*

Certaines méthodes de prévention des interblocages utilisent un classement des tampons. Les messages ne peuvent alors accéder qu'à des sous-ensembles de tampons, le reste lui est interdit. Ceci peut conduire à une utilisation à la fois inefficace et complexe de l'ensemble des tampons consacrés au routage.

Propriété 7.5 : Efficacité d'exécution. *Les actions entreprises pour éviter les interblocages ne doivent être au détriment ni de la capacité de calcul des processeurs ni de l'efficacité de communication du réseau ; en d'autres termes, ces actions doivent avoir un coût d'exécution négligeable. L'action du* **routeur** *doit être réalisée en un temps minimun.*

Propriété 7.6 : Minimisation de la longueur des routes. *Le délai de transfert d'un message entre le processeur source et le processeur destination est très dépendant du nombre de liens traversés, c'est-à-dire de la longueur de la route suivie. L'algorithme doit maintenir, autant que possible, les plus courts chemins.*

7.3 Les méthodes de lutte contre l'interblocage

Les techniques proposées dans la littérature pour résoudre le problème d'interblocage sont nombreuses, il est donc difficile de les passer toutes en revue et de les classifier. On peut cependant séparer les méthodes en deux grandes approches : celles qui agissent par *prévention* et celles qui agissent par *détection et guérison.*

7.3.1 Les techniques de détection-guérison

Les travaux qui appliquent cette technique aux problèmes d'interblocage dans le routage de messages[2] sont bien moins nombreux que ceux menés sur les techniques de prévention. Les caractéristiques générales des techniques de détection-guérison sont [CJS87, ChaYu87] :

- Aucune mesure n'est prise "a priori" pour éviter les interblocages.
- Un algorithme de détection des interblocages est mis en place en même temps que les procédures normales de routage que nous avons défini dans la section 6.3.
- Lorsqu'un interblocage est détecté, un mécanisme de guérison est activé pour l'éliminer.

Les principaux inconvénients de ces techniques sont :

- Elles n'accomplissent pas la proprieté 7.5 car elles requièrent l'exécution de procédures additionnelles et notamment de l'échange de messages additionnels.

[2]Ces techniques sont plutôt examinées dans le cadre des Bases de Données.

- Les algorithmes de détection sont complexes car ils se basent en général sur la détection des cycles étendus sur tout le réseau et ceci "pendant l'exécution". Outre l'obstruction sur les programmes utilisateurs, la détection même est compliquée pour les réseaux de taille importante.

- La détection implique normalement l'observation du comportement temporel des files d'attente, des "délais de garde" sont couramment utilisés pour vérifier l'activité dans les files. La détermination d'un délai de garde approprié à toute situation n'est pas simple ; ce délai devrait être fixé en fonction de l'activité réelle du réseau.

Nous considérons que les techniques de détection et guérison ne sont pas adaptées aux machines massivement parallèles.

7.3.2 Les techniques de prévention

Résoudre le problème d'interblocage correspond à établir un ensemble de routes pour l'acheminement de messages de telle façon qu'aucun cycle de demandes ne puisse se produire ; toutes les solutions se basent alors sur des méthodes définissant un *ordre d'utilisation des ressources*. Les approches suivies pour prévenir l'interblocage se retrouvent dans l'un des grands axes suivants :

Attribution permanente de tampons aux routes

Un tampon de stockage est réservé sur chaque nœud et pour chaque route. Il s'agit en effet de la virtualisation de la ressource mémoire ; les tampons ne sont plus une ressource partagée : aucun interblocage ne peut se produire. Le coût en espace mémoire est important et dépend de la taille du réseau.

C'est cette méthode qui est cependant à la base de celle proposé par Dally et Seitz dans [DaSe87] ; l'algorithme est basé sur le concept de graphe de dépendance entre liens (voir la section 7.4.1 de cette thèse), et le concept de *canaux virtuels* : un ensemble de canaux qui partagent un même lien physique mais où chaque canal a sa propre file d'attente. Les canaux virtuels sont totalement ordonnés par numérotation et l'algorithme de routage consiste simplement à faire qu'un message soit toujours routé suivant l'ordre décroissant des numéros des canaux virtuels.

La technique des canaux virtuels est appliquée par ses auteurs dans certains réseaux à topologie régulière. La création de canaux virtuels est réduite au minimun nécessaire pour maintenir le routage sans interblocages. Le coût mémoire peut donc être réduit par rapport à l'assignation sans restriction d'un tampon par

nœud et par route ; il reste néanmoins très important et dépendant de la taille et de la topologie du réseau. Il faut toutefois remarquer que bien que cette méthode soit utilisable pour le stockage-et-réexpédition, elle a été originellement proposée pour la technique *Wormhole*, que nous avons présenté dans la section 6.2 ; dans ce cas, les besoins de stockage sont fortement réduits : dans *Wormhole* on ne stocke que l'information minimale nécessaire à l'établissement de la connexion, c'est-à-dire la destination du message.

Vis-à-vis du routage par stockage-et-réexpédition, la méthode des canaux virtuels qui est reprise aussi dans [HiLu87], n'a que les propriétés 7.1, 7.5 et 7.6 ; elle est mieux adaptée à la technique *Wormhole*.

Restriction sur l'utilisation des tampons

Il s'agit d'éliminer les cycles de demande d'accès aux tampons par la définition d'un ordre dans son utilisation ; pour ce faire, l'ensemble des tampons d'un nœud est divisé en classes ; l'algorithme de prévention des interblocages consiste simplement à interdire l'accès de certains messages à certaines classes de tampons.

Plusieurs méthodes de classification des tampons ont été proposées ([GHKP78, GeKle80]). Les tampons d'un nœud sont divisés en K classes, où K correspond au diamètre du réseau. Un message qui arrive à un nœud intermédiaire à distance i du nœud source ne peut accéder qu'aux tampons de classe j tel que $j \leq i$. La même technique est reprise avec certaines variantes dans [AnTwi87], [Gopal85], [Shyam84] et [ToUll81].

Une technique un peu différente est proposée par Merlin et Schweitzer dans [MS80a], au lieu de classer les tampons, on construit un graphe orienté acyclique où les tampons sont les sommets. Les messages peuvent être transférés d'un tampon à un autre si et seulement si les deux tampons sont reliés par un arc dans le graphe.

Ces techniques les propriétés 7.1, 7.5 et 7.6 ; leur principal inconvénient est que la quantité de tampons dépend de la taille du réseau.

Construction de routes sur des graphes acycliques

Dans [Geler81], Gelernter propose une technique dans laquelle un graphe orienté et acyclique est construit sur le réseau original, les routes sont définies sur le graphe acyclique ce qui permet d'éviter les interblocages. Par cette méthode il est possible d'obtenir les propriétés 7.2, 7.3, 7.4 et 7.5. Le grand inconvénient de la méthode de Gelernter est qu'elle peut éliminer des messages : les propriétés

7.1 et 7.6 ne seraient donc pas assurées.

Cette idée de router sur des graphes acycliques est la base des méthodes de routage sans tables [SaKha85, LeTa87]. Elle est aussi proposée par Inmos pour les réseaux utilisant le transputer T9000 [Pount90], nous avons analysé la méthode dans la section 6.4.

Certains auteurs proposent le routage sur des graphes acycliques obtenus à partir de la construction d'un cycle sur le graphe original. L'idée consiste à construire un cycle hamiltonien (ou eulerien) s'il existe. Du point de vue du routage, le réseau est alors un anneau[3] et l'interblocage est éliminé soit en assurant qu'il aura toujours un tampon libre dans l'anneau soit en cassant le cycle par élimination d'un et un seul des liens qui le composent.

Cette méthode est utilisée par Roscoe [Rosc87] et reprise dans [GCST90] pour des réseaux de transputers organisés selon une topologie en grille à deux dimensions. Cette technique a les propriétés 7.1, 7.2, 7.3 et 7.4 ; son principal inconvénient est un allongement de la longueur des routes par rapport au plus court chemin. Pour éviter cela, il est possible d'utiliser, sous certaines conditions, des liens formant une corde dans l'anneau, les routes peuvent ainsi être raccourcies.

Le routage sur des graphes acycliques est à la base de deux algorithmes originaux de prévention d'interblocage développés au sein de l'équipe SYMPA. Le premier est basé sur la construction d'un arbre recouvrant et nous l'examinons en détail dans la section 7.5. Le second, basé sur la construction d'un cycle eulerien, a été publié dans [MMS90].

Marquage des messages par estampilles

Cette méthode est assimilable à celles proposées pour éviter les interblocages lors de la gestion de ressources dans les systèmes répartis[4], chaque message reçoit une estampille unique laquelle est utilisée lorsque des conflits d'accès aux tampons peuvent se produire. La méthode présentée dans [BBG87a] a les propriétés 7.1, 7.2, 7.3 et 7.4. L'inconvénient est celui des méthodes présentées dans le paragraphe précédent.

[3] Ou plusieurs anneaux indépendants, les messages pouvant passer d'un anneau à un autre selon les besoins du routage. C'est la méthode proposée dans [GCST90].

[4] Voir par exemple [CORNA81] ou [Krako87a].

7.4 Caractérisation de l'interblocage

Le caractère cyclique qui définit la possibilité d'existence d'un interblocage peut être exprimée de manières très diverses. En particulier, la fonction de routage définie dans la section 6.4 permet d'exprimer une relation de dépendance entre liens, laquelle est examinée dans le paragraphe suivant.

7.4.1 Graphe de relation de dépendance entre liens

Associé au graphe $RP = (P, L)$, muni de la fonction de routage R, il existe un graphe orienté $D = (L, E)$ dans lequel les sommets sont les liens du RP et les arcs sont des paires de liens connectés par la fonction R. Autrement dit, il existe un arc en D, orienté d'un sommet l_x^s vers un sommet l_y^d, si et seulement si, ce dernier est un successeur du premier dans une route quelconque définie par R. Les arcs représentent donc une relation de succession. Par la définition de R, le graphe D ne contient pas de cycle ayant deux liens ; tous les cycles sont de longueur supérieure à deux.

Un sous-graphe de D, qui correspond à la relation de dépendance entre liens d'une seule route entre deux processeurs quelconques, est nécessairement acyclique. Cependant, le graphe D peut contenir des cycles lorsque plusieurs routes ont de liens en commun (les routes ne sont pas disjointes).

Considérons le réseau de la figure 7.1a, et supposons qu'on construise l'ensemble de routes suivant :

$route_1(p_3, p_{12}) = l_1^3,\ l_1^4,\ l_2^5,\ l_2^9$

$route_1(p_{11}, p_6) = l_0^{11},\ l_0^8,\ l_1^4,\ l_1^5$

$route_1(p_2, p_7) = l_2^2,\ l_2^5,\ l_3^9,\ l_3^8$

$route_1(p_{10}, p_1) = l_3^{10},\ l_3^9,\ l_0^8,\ l_0^4$

Cet ensemble de routes est illustré dans la figure 7.1b ; le graphe D qui lui correspond est montré dans la figure 7.1c. On peut constater que D contient un cycle formé par la dépendance cyclique entre l'ensemble de liens : $\{l_1^4,\ l_2^5,\ l_3^9,\ l_0^8\}$.

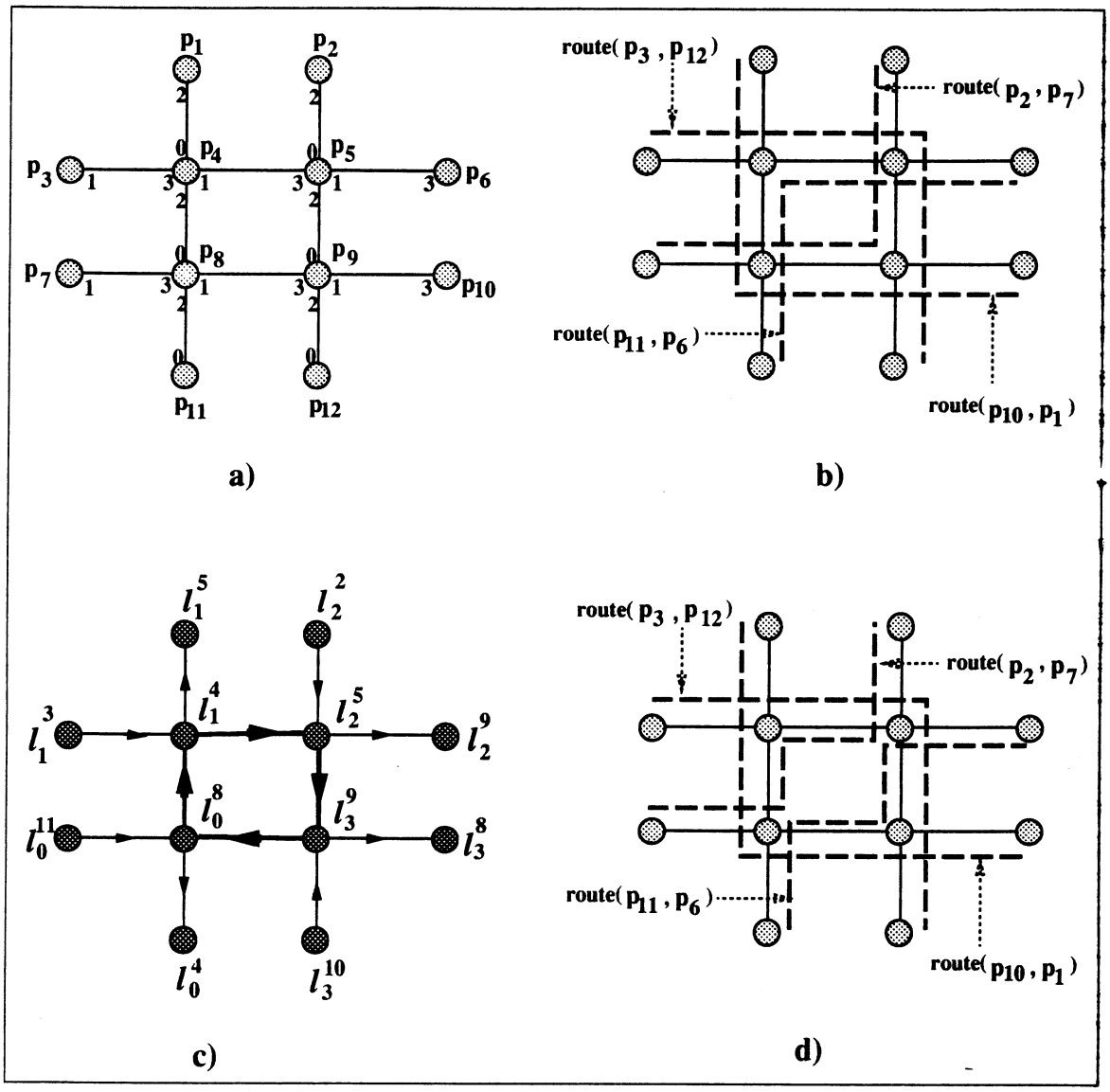


Figure 7.1: Illustration de la signification du graphe de dépendance entre liens. a) Le réseau de processeurs RP. b) Premier ensemble de routes définies par la fonction de routage qui produit un cycle dans le graphe D. c) Le graphe D associé aux routes illustrées dans b). d) Un ensemble de routes pour lequel le graphe D n'aura pas de cycles.

Supposons maintenant qu'on construit l'ensemble de routes suivant [5] :

$route_2(p_3, p_{12}) = l_1^3,\ l_1^4,\ l_2^5,\ l_2^9$

$route_2(p_{11}, p_6) = l_0^{11},\ l_1^8,\ l_0^9,\ l_1^5$

$route_2(p_2, p_7) = l_2^2,\ l_3^5,\ l_2^4,\ l_3^8$

$route_2(p_{10}, p_1) = l_3^{10},\ l_3^9,\ l_0^8,\ l_0^4$

Les routes sont illustrées par les figures 7.1d, on peut constater que ces routes n'ont aucun lien commun ; le graphe D ne peut alors contenir aucun cycle.

Dally et Seitz [DaSe87] montrent que tout interblocage se traduit nécessairement par un cycle dans le graphe D ; nous utilisons donc le résultat suivant :

> *"la fonction de routage R sur le réseau RP est sans interblocage si et seulement si le graphe D est sans cycles."*

7.4.2 Un exemple d'interblocage

Supposons que dans le réseau de la figure 7.1a, des messages sont échangés de la façon suivante :

p_3 envoie des messages $m_{3,12}$ vers p_{12} par $route_1(p_3, p_{12})$

p_2 envoie des messages $m_{2,7}$ vers p_7 par $route_1(p_2, p_7)$

p_{10} envoie des messages $m_{10,1}$ vers p_1 par $route_1(p_{10}, p_1)$

p_{11} envoie des messages $m_{11,6}$ vers p_6 par $route_1(p_{11}, p_6)$

les processeurs envoient leurs messages indépendamment les uns des autres, et aucune hypothèse n'est faite quant au débit des messages. Si à un instant donné la situation des files d'attente est la suivante : (fl_i^j dénote la file du lien l_i^j)

fl_2^5 est pleine de messages $m_{2,7}$ utilisant $route_1(p_2, p_7)$

[5] Le lecteur peut noter que la longueur des routes est maintenue par rapport au premier ensemble.

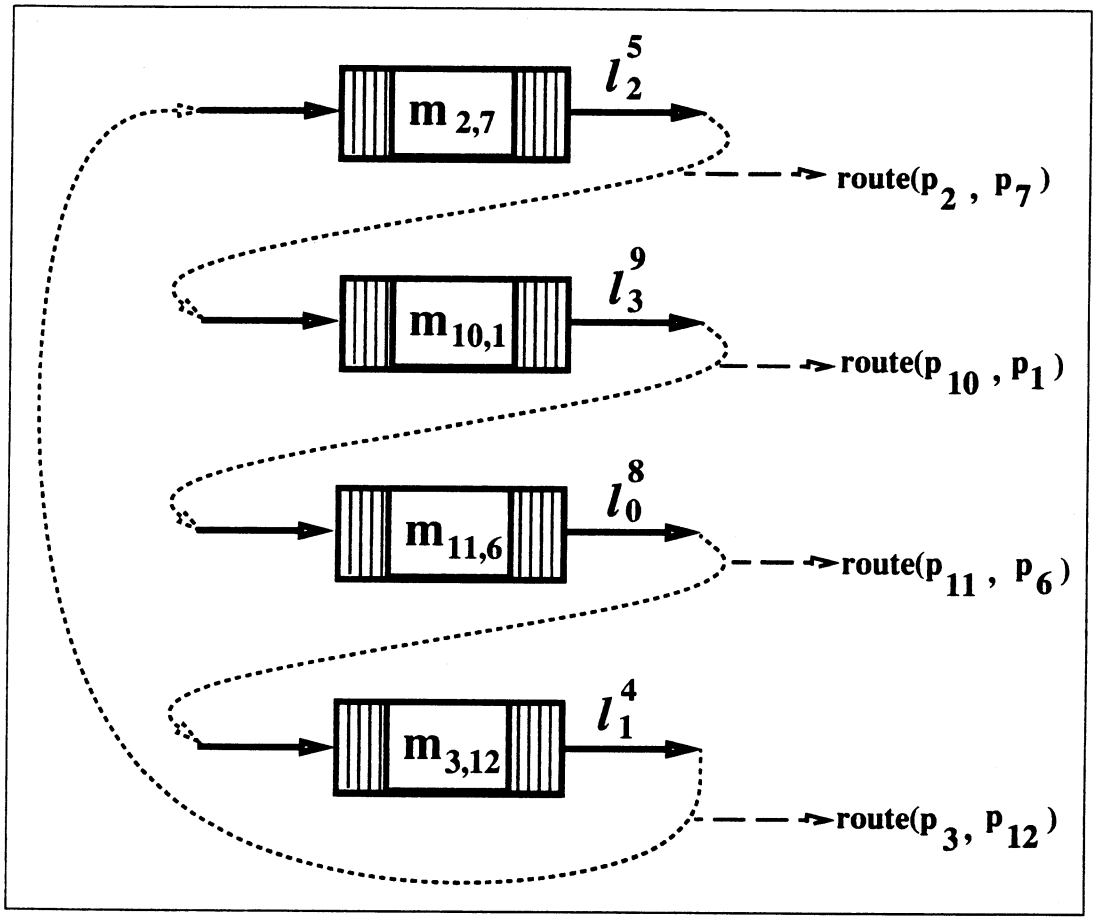


Figure 7.2: Exemple d'interblocage

fl_3^9 est pleine de messages $m_{10,1}$ utilisant $route_1(p_{10}, p_1)$

fl_0^8 est pleine de messages $m_{11,6}$ utilisant $route_1(p_{11}, p_6)$

fl_1^4 est pleine de messages $m_{3,12}$ utilisant $route_1(p_3, p_{12})$

tous les messages se trouvent à deux "pas" de distance de leur destination, où ils seraient "consommés", autrement dit, où ils quitteraient le réseau de routage et libéreraient des places dans les files ; aucun message ne peut avancer car, pour chacun d'eux, la prochaine file d'attente en aval de la route est pleine. La figure 7.2 représente cette situation.

7.5 Les fondements d'un nouvel algorithme

Le résultat de la section 7.4.1 est le point de départ de l'obtention d'une fonction de routage sans interblocage. En pratique le problème se pose de la façon suivante :

> *"pour éviter des interblocages dans le routage des messages il faut construire les routes de façon à éviter l'existence de cycles dans le graphe D de dépendance entre liens."*

Une condition nécessaire pour obtenir le routage sans interblocage est que lorsqu'un message arrive au processeur destination, il soit "consommé" dans un temps fini. Ceci impose une condition sur la couche des protocoles : il doit y avoir toujours un processus qui prend les messages en provenance du routeur.

Nous allons présenter maintenant un algorithme de routage sans interblocage qui a les propriétés 7.1, 7.2, 7.3, 7.4 et 7.5. La propriété 7.6 sera examinée dans la section 7.8, où nous comparons notre algorithme avec ceux ayant quelques propriétés communes. Examinons d'abord les approches qui peuvent conduire à une solution basée sur la construction d'un graphe D acyclique.

Les réseaux à connectique régulière

Exhiber un algorithme de routage réalisant un graphe D acyclique pour des réseaux à connectique régulière s'avère en général une tâche aisée et peut conduire à des solutions optimales du point de vue de l'espace mémoire, la longueur des routes et le temps d'exécution. Ce type d'algorithme n'a pas la propriété 7.2 ; le routage est normalement restreint par l'algorithme et il s'applique à des réseaux ayant des topologies particulières.

Un bon exemple est l'algorithme *e-cube* utilisé pour les hypercubes de dimension k : chaque nœud est étiqueté par n_d où d est un numéro binaire à k chiffres. L'algorithme *e-cube* achemine les messages par ordre décroissant des dimensions : par exemple, un message qui arrive au nœud n_d ayant comme destination le nœud n_l est réexpédié par un lien l_i^d, où i est la position du bit le plus significatif pour laquelle d et l diffèrent. Les messages étant toujours routés par ordre décroissant des dimensions aucun cycle ne peut se produire et l'algorithme est sans interblocage.

La fonction réalisée par l'algorithme *e-cube* est suffisamment simple pour être calculée localement sur chaque processeur et sans besoin de tables de routage ; c'est la base de la méthode utilisée par le Torus Routing Chip.

Les réseaux à connectique acyclique

Le routage sur un réseau à topologie acyclique est naturellement libre d'interblocages lorsque chaque route est acyclique[6].

La structure la plus représentative des réseaux acycliques est l'arbre. Le routage sur un arbre a été proposé, par exemple, dans [MS80a]. La seule règle de routage consiste à interdire qu'un message puisse visiter plus d'une fois le même processeur ; autrement dit, une route ne peut contenir plus d'une fois le même lien. L'algorithme de routage est très simple et se réalise sans besoin de tables de routage : pour acheminer un message d'un processeur p_s à un processeur p_d, il suffit d'emprunter la suite de liens ascendants et la suite des liens descendants reliant les deux processeurs à leur ancêtre commun dans l'arbre[7]. L'espace mémoire nécessaire au routage se réduit à un seul tampon par lien dans chaque nœud.

Lorsque cet algorithme s'applique à un réseau qui a une topologie comportant des cycles, il présente les caractéristiques suivantes :

1. L'ensemble des liens effectivement utilisés par le routage n'est qu'un sous-ensemble de tous les liens présents dans le réseau d'origine. La capacité de communication du réseau est donc sous-utilisée.

2. Les liens des niveaux proches de la racine de l'arbre peuvent être rapidement saturés.

3. Les routes ne sont pas toujours construites en utilisant le plus court chemin.

L'algorithme basé sur la notion de *routage par intervalles* utilise aussi la structure arborescente comme support de routage ; mais il essaie d'éliminer les inconvénients indiqués ci-dessus. En effet, dans [LeTa87] il est montré comment l'algorithme s'applique à certains réseaux dont la topologie n'est pas acyclique (grilles à deux dimensions et tore). Les inconvénients de cet algorithme ont été soulignés dans la section 6.4.

Le support théorique

Outre les inconvénients déjà énoncés, l'inconvénient commun aux deux classes de solutions présentées ci-dessus est qu'elles ne sont pas applicables (sans modifications) aux réseaux à topologie irrégulière.

Nous proposons de remédier à ces inconvénients par un algorithme ayant comme

[6] Aucun nœud ne peut être visité plus d'une fois par le même message.

[7] Les termes "lien ascendant" et "lien descendant" sont définis dans la section 7.6.

support un arbre recouvrant le graphe RP, mais qui utilise les liens de communication de RP ne faisant pas partie de l'arbre. L'algorithme se base sur les résultats suivants :

Res 7.1 *Par définition, les liens formant un cycle dans le graphe D forment nécessairement un cycle dans le graphe RP.*

La réciproque n'est pas vraie : un cycle dans RP n'induit pas, par lui-même, un cycle dans D. Un cycle est produit par les routes définies par R dans RP.

Res 7.2 *Le routage dans tout réseau pour lequel RP est acyclique, est sans interblocage indépendamment de la fonction de routage.*

Res 7.3 *Un arbre est par définition acyclique. Le graphe D associé à un arbre est, en vertu du résultat* **Res 7.1**, *lui aussi acyclique.*

Res 7.4 *En vertu des résultats* **Res 7.2** *et* **7.3**, *le routage dans un arbre est sans interblocages.*

Res 7.5 *Tout graphe connexe admet un arbre recouvrant.*

7.6 Formalisation de l'algorithme

Considérons d'abord les définitions et notations suivantes :

$AR = (P, L_a)$: est l'arbre recouvrant du graphe RP ; les sommets de AR sont ceux de RP. L'ensemble des arcs de AR est un sous-ensemble des arcs de RP. La racine de AR est le processeur p_r. Comme RP est connexe, en vertu du résultat **Res 7.5**, il admet un arbre recouvrant.

$EL = L - L_a$: EL[8] est l'ensemble des liens de communication du réseau représenté par RP et qui ne sont pas dans l'arbre AR. $EL = \emptyset \Longleftrightarrow AR = RP$.

$niv(\alpha)$: est le niveau d'un processeur p_α dans AR, qui correspond à la distance qui sépare p_α de p_r. Nous considérons $niv(r) = 0$.

$l_{\alpha,\beta} = (l_x^\alpha, l_y^\beta)$: est un élément de l'ensemble L reliant les nœuds p_α et p_β.

$sa(\alpha)$: dénote le sous-arbre de racine p_α.

$l_{\alpha,\beta} \xrightarrow{D} l_{\beta,\gamma}$: dénote l'existence d'un arc entre $l_{\alpha,\beta}$ et $l_{\beta,\gamma}$ dans D.

$l_{\alpha,\beta}$: est dit arc (ou lien) *ascendant* si $niv(\alpha) > niv(\beta)$.

[8] EL provient de ***Extra-Liens***, nom que nous avons donné originellement à l'algorithme.

$l_{\alpha,\beta}$: est dit arc (ou lien) *descendant* si $niv(\alpha) < niv(\beta)$.

$l_{\alpha,\beta}$: est dit arc (ou lien) *horizontal* si $niv(\alpha) = niv(\beta)$.

Nous avons déjà dit que l'algorithme que nous proposons utilise comme base l'arbre recouvrant, mais l'idée est d'utiliser tous les liens du réseau original, c'est-à-dire, tout l'ensemble L. Lorsque $EL \neq \emptyset$, le rajout successif des éléments $e \in EL$ au graphe AR conduit à RP. Cependant, un graphe $AR' = (P, L')$ où $L' = (L_a \cup e)$ contient nécessairement un cycle. Ce cycle, comme l'énonce le résultat **Res 7.1**, induit des interblocages potentiels ; autrement dit, la formation de cycles est maintenant possible dans le graphe D' associé au nouveau graphe AR'.

Les cycles formés dans D' sont néanmoins aisèment caractérisés à l'aide des proprietés de l'arbre recouvrant. L'arbre n'est donc qu'un moyen permettant de caractériser l'ensemble de cycles du graphe D, associé au graphe RP, qu'il faut éliminer pour obtenir un routage sans interblocage. Les liens de l'ensemble EL ne peuvent donc être utilisés librement pour le routage des messages : certaines règles doivent être introduites ; l'algorithme de routage n'est que la concrétisation de ces règles.

L'arbre recouvrant RP est construit en largeur d'abord et nous supposons qu'entre deux processeurs ne peut exister qu'un seul lien, d'où les propriétés suivantes :

Propriété 7.7 : *Il ne peut exister un lien $l_{\omega,\alpha} \in EL$ connectant deux nœuds se trouvant sur la même branche de l'arbre ; en d'autres termes, α et β sont des racines des sous-arbres $sa(\alpha)$ et $sa(\beta)$ ayant la propriété 7.9 énoncée ci-dessous.*

Propriété 7.8 : *Pour tout lien $l_{\omega,\alpha} \in EL$ on a la relation $\mid niv(\omega) - niv(\alpha) \mid \leq 1$*

Propriété 7.9 : *Soit $p_\alpha \in P$ et $p_\beta \in P$. Les sous-arbres correspondants $sa(\alpha)$ et $sa(\beta)$ n'ont aucun nœud commun dans les deux cas suivants :*

- $niv(\alpha) = niv(\beta)$
- $niv(\alpha) \neq niv(\beta)$ *et* $\alpha \notin sa(\beta)$ *et* $\beta \notin sa(\alpha)$

Propriété 7.10 : *Soit $p_\alpha \in P$ et $p_\beta \in P$, lorsque $\alpha \neq \beta$ et $\alpha \neq r$ et $\beta \neq r$, α et β ont un ancêtre commun unique que nous appelons γ. Les sous-arbres $sa(\alpha)$ et $sa(\beta)$ appartiennent au sous-arbre $sa(\gamma)$.*

7.6.1 Les règles de routage

Soit AR' le graphe obtenu par adjonction d'un élément $l_{\alpha,\beta} \in EL$. AR' contient alors un cycle élémentaire. Les règles de routage seront déduites à partir de la caractérisation de tous les cycles pouvant se produire à cause de $l_{\alpha,\beta}$.

Cycle 7.1 : *Supposons que* $niv(\alpha) \neq niv(\beta)$ *et sans perte de généralité supposons que* $niv(\alpha) > niv(\beta)$. *Un cycle contenant* $l_{\alpha,\beta}$ *contient nécessairement un lien* $l_{\omega,\alpha}$ *pour lequel* $\omega \neq \beta$ *et* $niv(\omega) = niv(\beta)$.

Démonstration : Par la propriété 7.7, $\alpha \notin sa(\beta)$. Comme $niv(\alpha) > niv(\beta)$, il existe nécessairement un nœud $\omega \neq \beta$ tel que $\alpha \in sa(\omega)$. Par la propriété 7.8, $niv(\beta) = niv(\alpha) - 1$ et comme $\beta \neq r$ il existe nécessairement un nœud au même niveau que β qui est l'ancêtre direct de α, ce nœud est donc ω. Par la propriété 7.9, $sa(\omega)$ et $sa(\beta)$ n'ont aucun nœud en commun, aucun autre cycle ne peut exister.

Règle de routage R1 : Un cycle de type 7.1 implique qu'un cycle dans D contient nécessairement l'arc $l_{\omega,\alpha} \xrightarrow{D} l_{\alpha,\beta}$, la règle de routage consiste donc à éviter qu'un lien descendant soit suivi d'un lien ascendant.

Cycle 7.2 : *Supposons que* $niv(\alpha) = niv(\beta)$. *Un cycle contenant* $l_{\alpha,\beta}$ *contient nécessairement deux liens,* $l_{\beta,\omega}$ *et* $l_{\alpha,\eta}$ *pour lesquels on a :*

- $\beta \in sa(\omega)$ *et* $\alpha \in sa(\eta)$
- $niv(\beta) > niv(\omega)$ *et* $niv(\alpha) > niv(\eta)$

Démonstration : Comme p_α et p_β sont au même niveau, par les propriétés 7.9 et 7.10, un cycle doit nécessairement contenir des arcs partant de α et β vers son ancêtre commun, ces sont les arcs $l_{\alpha,\eta}$ et $l_{\beta,\omega}$. Sans perte de généralité on peut faire $\omega = \eta = \gamma$ l'ancêtre commun de α et β, le cycle est donc formé par α, β et ω.

Règle de routage R2 : Un cycle de type 7.2 implique qu'un cycle dans D contient nécessairement l'arc $l_{\alpha,\beta} \xrightarrow{D} l_{\beta,\omega}$. La règle de routage consiste donc à éviter qu'un lien horizontal soit suivi d'un lien ascendant.

Les cycles 7.1 et 7.2 ont un lien et un seul qui fait partie de l'ensemble EL. Lorsque tout l'ensemble EL est rajouté à l'arbre, des cycles contenant plus d'un élément de l'ensemble EL peuvent se produire.

Lorsqu'un cycle contient au moins un lien de l'arbre, on se ramene à l'un des deux types de cycles présentés ci-dessus, le lien appartenant à l'arbre introduit

nécessairement un changement de niveau et le cycle contient donc nécessairement l'une des deux dépendances énoncées par les règles de routage R1 et R2.

Soit un cycle $\mathcal{C}$ qui ne contient aucun lien de l'arbre et soit le lien $l_{\alpha,\beta}$ qui appartient à la fois à EL et à $\mathcal{C}$, l'un des deux cas suivants peut se présenter :

1. Lorsque $niv(\alpha) \neq niv(\beta)$, le cycle $\mathcal{C}$ contient des nœuds aux niveaux différents et on revient nécessairement à l'une des règles de routage R1 ou R2.

2. Le cycle ne contient que des liens horizontaux, c'est-à-dire : $\forall\ l_{\alpha,\beta} \in \mathcal{C}$, $niv(\alpha) = niv(\beta)$. Les règles de routage R1 et R2 ne sont plus applicables, d'où le besoin de la règle suivante.

Règle de routage 3 : Un cycle ne contenant que des liens horizontaux implique que le cycle correspondant dans D contient nécessairement la succesion suivante :

$l_{\alpha,\beta} \xrightarrow{D} l_{\beta,\omega} \xrightarrow{D} l_{\omega,\eta} \xrightarrow{D} \ldots \xrightarrow{D} l_{\gamma,\alpha}$.

Dans le cas le plus strict, il faudrait éviter qu'un lien horizontal soit suivi d'un lien horizontal, mais il suffit d'éviter la formation d'un cycle contenant seulement des liens horizontaux en éliminant un arc quelconque du graphe D faisant partie de $\mathcal{C}$.

7.7 Complexité de l'algorithme

L'algorithme de calcul des tables de routage sans interblocage consiste à :

1. Calculer l'arbre recouvrant et marquer chaque sommet avec le niveau correspondant. Chaque lien est donc implicitement marqué ascendant, descendant ou horizontal.

2. Calculer les routes entre toute paire de processus. Il s'agit du calcul du plus court chemin entre chaque paire de processus du graphe RP, en tenant compte des restrictions imposées par les règles de routage.

Ces deux étapes de calcul correspondent à des algorithmes très connus, d'où notre décision de ne pas les présenter ici. La seule modification à introduire aux algorithmes est la prise en compte des règles de routage définies dans la section 7.6, ce qui ne change en rien la complexité des algorithmes originaux.

Il existe deux formes possibles de mise en œuvre, la première est séquentielle et la deuxième est parallèle et utilise le réseau cible.

Le complexité d'un algorithme séquentiel est déterminée fondamentalement par le calcul des plus courts chemins dans le réseau original muni des règles de routage. Cela correspond au calcul de la fermeture transitive du graphe RP, qui a une complexité de $O(N^3)$, de tels algorithmes sont présentés dans [AHU74].

La complexité d'un algorithme parallèle est déterminée par la plus grande distance entre deux paires de sommets, ce qui est couramment défini comme le *diamètre* du réseau.

7.8 Evaluation de l'algorithme

L'algorithme que nous avons proposé, et que nous appellerons EL, a les propriétés 7.1, 7.2, 7.3, 7.4 et 7.5. Cependant, la propriété 7.6 mérite une évaluation car l'utilisation de l'arbre recouvrant comme structure base pour la détermination des routes ne permet pas d'assurer l'utilisation du plus court chemin. L'étude suivante a pour objectif d'évaluer les caractéristiques de l'algorithme en ce qui concerne la longueur des routes.

Facteur d'élongation des routes

Une mesure de l'efficacité de l'algorithme EL, en ce qui concerne la longueur des routes, est le *facteur d'élongation*, qui a été défini dans [PeUp89] et que nous reprenons ici.

Définition. Soit $p_a \in P$ et $p_b \in P$. Soit $d_{RP}(p_a, p_b)$ la distance de p_a à p_b, qui correspond en réalité à la plus courte distance entre les processurs p_a et p_b dans le graphe RP. Soit r la route de p_a à p_b obtenue par l'algorithme EL. Le *facteur d'élongation* de r est défini par :

$fer = max_{p_a,p_b \in P}\{\frac{\mathcal{L}(r)}{d_{RP}(p_a,p_b)}\}$.

Définition. Soit $G = (V, E)$ un graphe quelconque. Prenons $x \in V$ et soit $r_G(x) = max\{d_G(x, y) \mid y \in V\}$; un nœud $c \in V$ est un *centre* du graphe G si pour tout $x \in G$, $r_G(c) \leq r_G(x)$.

Définition. Si c est le centre de G, alors $r_G(c) = r(G)$ est le *rayon* de G et pour tout $x, y \in V, d_G(x, y) \leq 2r(G)$.

Santoro et Khatib [SaKha85] ont démontré que si l'arbre recouvrant d'un graphe G quelconque a comme racine le centre de G, et l'arbre est tel que la distance

entre la racine et n'importe quel nœud soit la plus courte, alors la plus grande distance séparant deux nœuds dans l'arbre est inférieur ou égale au rayon du G.

Le résultat est directement applicable à l'algorithme EL : soit $p_c \in P$ le centre de RP et $r_{RP}(p_c)$ le rayon de RP, si p_c est la racine de AR alors pour tout $p_a, p_b \in P$, $d_{AR}(p_a, p_b) \leq r_{RP}(p_c)$.

Théorème 7.1 *Le facteur d'élongation des routes de l'algorithme EL, appliqué à un réseau RP quelconque, est inférieur ou égal $2r(RP)$ si la racine de AR est le centre de RP.*

Démonstration. Considérons deux nœuds quelconques $p_a \in P$ et $p_b \in P$, tels que $p_a \neq p_b$. Indépendamment de la distance $d_{RP}(p_a, p_b)$ qui sépare les nœuds dans RP, lorsque la racine de AR est le centre de RP, la distance maximale pouvant les séparer dans AR est $d_{AR}(p_a, p_b) \leq 2r(RP)$. Dans le cas le plus défavorable $d_{RP}(p_a, p_b) = 1$ et $\mathcal{L}(r) = d_{AR}(p_a, p_b)$, le résultat est alors immédiat.

7.9 Résultats expérimentaux

Nous avons écrit une version séquentielle de l'algorithme, laquelle est actuellement utilisée aussi bien dans le prototype de Parx, que nous sommes en train d'évaluer au sein de l'équipe SYMPA, que dans le prototype du système évalué dans le projet **ESPRIT SUPERNODE II**. Cette version permet de calculer les tables de routage sans interblocages pour les diverses topologies du réseau pouvant être utilisées au cours des évaluations.

La seule mesure qui nous intéresse est le facteur d'élongation des routes. Pour ce faire, on a calculé la distance $d_{RT}(p_a, p_b)$ pour tout $p_a \in RT$, $p_b \in RT$. La longueur des routes déterminées par l'algorithme a été aussi calculée.

Bien que l'algorithme soit applicable à n'importe quelle topologie, la validation de nos résultats n'est possible que par l'application de l'algorithme sur les topologies les plus connues.

Hypercubes binaires d'ordre k:

Rappelons qu'un k-cube a 2^k nœuds, et que dans le cas des transputers k est limité à quatre[9].

[9]Sauf si l'on introduit du multiplexage matériel sur les liens pour obtenir une connectivité plus grande.

Le facteur d'élongation, obtenu des mesures réalisées pour les hypercubes binaires à $k = 2, 3, 4$ et 8, fut toujours un, c'est-à-dire que toutes les routes sont construites sur les plus courts chemins.

Grilles à deux dimensions

Une grille contient $n \times m$ nœuds organisés en n colonnes et m lignes ; elle n'exige qu'un maximum de quatre liens indépendamment des valeurs de n et de m, par conséquent, avec des transputers on peut construire des grilles sans limitation de taille.

Dans nos essais, nous avons fait $n = m$, et nous avons fait varier la valeur de n entre deux et dix. Le résultat obtenu est celui des hypercubes, le fer mesuré est toujours unitaire. L'algorithme est alors optimum, dans ce sens, aussi pour les grilles.

Dans une grille, tout nœud n'est pas un *centre*, le choix de la racine de l'arbre recouvrant est alors à tenir en compte. Nous avons donc essayé l'algorithme en prennant trois types de racine : un nœud dans un coin de la grille (qui n'est connecté que par deux liens), un nœud de la péripherie (qui n'est connecté que par trois liens) et un nœud du centre de la grille (qui est connecté par ses quatre liens). Dans tous les cas, le fer mesuré fut l'unité.

Grilles à deux dimensions rebouclées : *tore*

Le *tore* est une grille dans laquelle, pour chaque file, chaque nœud de la colonne la plus à gauche est connecté au nœud de la colonne la plus à droite. Le même principe s'applique aux colonnes par rapport aus files.

Nous avons mesuré le fer pour $n = 4, 5, 6, 8$ et 10. Si $n = 4$ le tore est un 4-cube et le fer est un. Pour les tores de taille plus grande, le fer n'est plus l'unité pour toutes les routes, les résultats sont résumés dans la table 7.1 :

n	*rayon*	$fer > 1$	*max.fer*
5	4	4/100	1.5
6	6	2/100	2.0
8	8	9/1000	3.0
10	10	4.6/1000	4.0

Table 7.1: Facteur délongation de routes dans le *tore*.

La colonne $fer > 1$ indique la quantité de routes qui présentent un facteur d'élongation supérieur à un et la colonne *max.fer* indique le plus grand fer. Nous constatons que le nombre de routes qui ont un fer supérieur à un est très réduit et que le fer maximun est loin de la borne supérieure fixée par le théorème 7.1.

Un travail qui reste à faire consiste à démontrer théoriquement les résultats expérimentaux montrés ici, en particulier ceux qui concernent les hypercubes et les grilles. On pourrait aussi étudier de possibles améliorations aux résultats concernant le *tore*.

7.10 Conclusion

Le problème d'interblocage est sans doute l'un des plus importants dans le routage des messages. Il existe de nombreuses solutions, cependant, aucune ne satisfait tous les besoins et notamment ceux particuliers aux architectures multiprocesseurs sans mémoire commune à réseau d'interconnexion modifiable et grand nombre de processeurs.

Nous avons proposé un algorithme original, qui a comme caractéristique principale d'être applicable à n'importe quelle topologie du réseau d'interconnexion. Etant basé sur l'élimination des cycles par un arbre recouvrant, le besoin de tampons mémoire pour le routage est réduit à un tampon par lien et par processeur, et cela indépendamment de la taille et de la topologie du réseau. Condition qui était imposée comme strictement nécessaire pour les architectures visées. Le seul coût de l'algorithme est l'allongement des routes ; nos mesures indiquent cependant que ce coût est nul pour les grilles et les hypercubes, et très faible pour les tores.

Reste seulement à rappeler que l'élimination de l'interblocage au niveau routage ne préserve pas de leur apparition dans les autres couches logiciels, notamment celles de l'interprétation des protocoles, que nous examinerons dans le chapitre suivant, et au niveau utilisateur, sujet qui est hors de la portée de nos travaux.

Chapitre 8

Interprétation de protocoles

8.1 Introduction

L'objectif de la couche d'interprétation de protocoles est de permettre la communication entre processus, indépendamment de leur localisation et éventuellement par des protocoles différents. Cette couche réalise les fonctionnalités liées à la communication au niveau du noyau de Parx. Les fonctionnalités du noyau concernant la gestion de processus sont traitées dans [Langue91].

On propose une architecture générique de la couche d'interprétation qui s'applique à tout protocole. Ceci permet de maintenir l'architecture et de ne changer que les fonctionnalités particulières de chaque protocole, des protocoles peuvent être rajoutés ou éliminés sans qu'il y ait besoin de modifier les autres couches du système.

La couche d'interprétation des protocoles accède aux mécanismes de routage en utilisant l'interface simple par files d'attente. Il faut se rappeler que lorsqu'un message arrive à destination, il est passé par le routeur aux protocoles, et qu'une condition nécessaire pour que le routage soit sans interblocage est que les protocoles soient toujours prêts à accepter un message du routeur. Cette condition signifie concrètement que la fonction de réception d'un protocole ne doit pas être soumise à une quelconque condition à durée de réalisation indéfinie. Lorsqu'un message est accepté par un protocole, il quitte la couche de routage libérant la place qu'il utilisait dans le réseau de routage pendant son acheminement.

Le reste de ce chapitre présente d'abord l'architecture générique et ensuite les protocoles considérés par Parx.

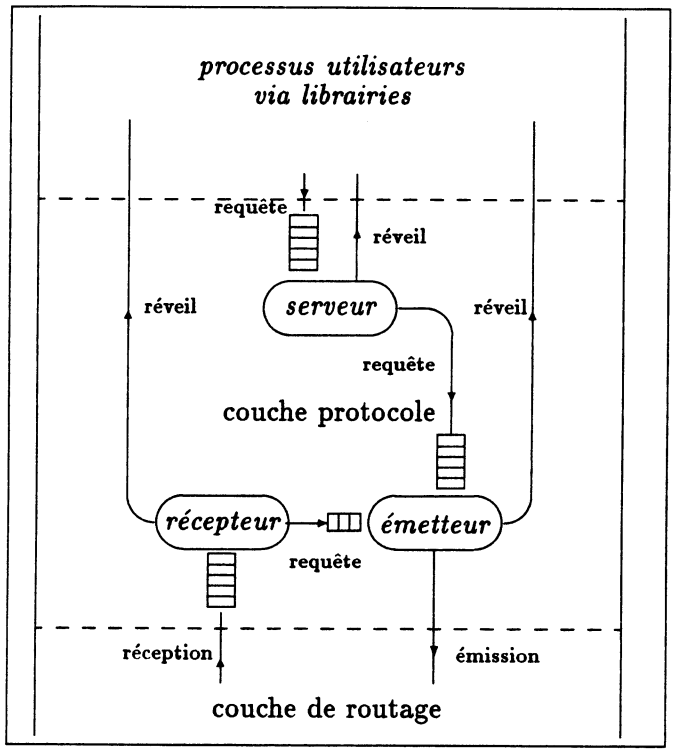


Figure 8.1: Structure de la couche protocole

8.2 Architecture de la couche de protocoles

Le comportement de chaque protocole est encapsulé dans un ensemble de trois processus. Le *récepteur* reçoit les messages du réseau , le *serveur* interface le noyau avec l'utilisateur, et l'*émetteur* transmet les messages sur le réseau. Tous les protocoles sont construits par le même ensemble de processus. La figure 8.1 est une illustration de l'interaction entre ces trois processus. Le même ensemble est dupliqué pour chaque protocole et il est possible d'en dupliquer aussi pour un même protocole. Chaque processus est en fait un automate réalisant un ensemble de fonctions qui constituent les tâches élémentaires d'un protocole.

Chaque processus s'occupe d'un type d'interaction et offre une interface appropriée à celui qui demande ses services. Les interactions sont analysées ci-dessous.

Interaction *utilisateur-noyau de communication* **réalisée par le** *serveur*

L'utilisateur du noyau peut être soit directement un programme application soit, et plus couramment, une couche supérieure du système d'exploitation. Quel que soit le cas, une requête de service doit être soumise au serveur. Celui-ci vérifie la validité et active la fonction demandée par la requête, qui est chargée de démarrer le protocole de communication. Par exemple, comme nous le verrons pour le protocole occam, elle requiert de l'émetteur du protocole de l'envoi d'un paquet de demande de rendez-vous, adressée au processus correspondant dans la communication.

Interaction *routage-protocoles* **réalisée par le** *récepteur*

Le récepteur prend un paquet de la file qui l'interface avec le routeur et récupère l'information nécessaire pour déterminer la fonction demandée, vérifie la validité et l'exécute. Les fonctions activées par ce processus sont essentiellement chargées de l'interprétation des données reçues et de l'exécution des actions qui en résultent ; par exemple, stocker les données, solliciter de l'émetteur l'envoi d'un acquittement, etc.

Interaction *protocoles-routage* **réalisée par l'***émetteur*

En fonction des requêtes qui lui auront été soumises par les processus serveur et récepteur, l'émetteur active les fonctions chargées de construire un paquet et de le soumettre au routeur pour acheminement vers le processeur destination. L'émetteur a deux entrées, donc deux files d'attente. Normalement, les requêtes en provenance du récepteur seront prioritaires afin de privilégier la continuité des actions des protocoles en cours. La priorité doit cependant être contrôlée et modulée dans le but d'éviter la famine en ce qui concerne les requêtes en provenance du serveur.

8.2.1 Les files d'attente

Les processus récepteur et serveur peuvent déposer des requêtes d'émission de messages dans une file d'attente destinée à l'émetteur. Le récepteur lit les messages d'une file d'attente où sont déposés les messages en provenance du réseau. Le serveur dispose aussi d'une file d'attente pour les requêtes des utilisateurs.

Les files d'attente peuvent être bloquantes ou non bloquantes, et avec différentes stratégies de retrait d'un élément :

1. Premier arrivé-premier servi : c'est la politique la plus équitable, elle doit être utilisée pour éviter la famine chaque fois qu'on n'a pas besoin d'établir

des priorités.

2. File d'attente multi-niveaux sans priorité : chaque niveau correspond à une classe de client. L'ensemble des premiers clients de chaque niveau est d'abord servi. C'est une variation de la politique précédente destinée à fournir un service équitable pour chaque niveau.

3. File d'attente multi-niveaux avec priorité : la file est encore multi-niveaux, mais une priorité est associée à chaque niveau. Un nombre p, paramètre de création de la file, de clients du niveau de plus forte priorité sont traités avant qu'un client d'une priorité plus faible le soit.

D'un point de vue pratique, la gestion de la file d'attente doit être simplifiée au maximun, une file de type premier arrivé-premier servi est à la fois simple à gérer et évite la famine, les files multi-niveaux avec priorités doivent s'utiliser seulement si leur comportement est mieux adapté aux besoins d'un protocole. Une étude sur la manipulation des files est présentée dans l'annexe A.

Désignation et utilisation des protocoles

Chaque protocole peut être décrit par un ensemble de couples (*numéro de requête, traitement associé*). Le numéro de requête, contenu dans l'*en-tête* du paquet, permet d'identifier et d'activer le traitement à lui faire subir. Les couples peuvent être subdivisés en trois ensembles, respectivement reconnus par le récepteur, le serveur et l'émetteur.

Cette désignation permet la modification dynamique des associations numéro de requête-traitement. A la différence des "*streams*" d'Unix, qui permettent d'*empiler* et de *dépiler* des traitements à appliquer à un *flot de données*, notre désignation permet de *modifier* la réponse du protocole à un *type de requête* donné.

Déroulement général d'une communication

Lorsque l'utilisateur réalise une communication, il appelle une fonction du système qui crée une requête puis active le protocole. Les automates du protocole exécutent les actions nécessaires à l'établissement de la communication, puis en fonction de l'état du correspondant, continuent ou abandonnent la requête dans l'attente d'une synchronisation.

Pendant toute la durée de la communication, la requête doit être conservée dans le descripteur du processus ayant initié la communication[1] où on peut la

[1]Lors de la mise en œuvre, il est possible par exemple, d'utiliser l'espace de travail du processus qui est bloqué dans l'appel au noyau via une fonction du type `trap()`.

récupérer chaque fois qu'un nouvel élément permet de faire avancer la communication. Lorsque celle-ci est terminée, l'automate qui exécute la dernière action doit réveiller le processus utilisateur.

8.3 Les protocoles considérés par Parx

Dans la section 3.5.1 nous avons examiné les formes d'interaction entre processus. Deux modèles d'interaction, par données partagées et par échange de messages, ont été discutés. Le modèle de Parx n'offre de possibilité de communication par données partagées qu'entre les *threads* d'une même *tâche* sur le même processeur physique. Aucun support particulier, mis à part les mécanismes de synchronisation usuels sur les mono-processeurs, n'est proposé pour ce modèle d'interaction.

Concernant l'interaction par échange de messages, modèle de base de Parx, nous avons retenu : un protocole d'accès à la mémoire à distance, un protocole de rendez-vous point à point synchrone, un protocole client-serveur plusieurs vers un, et un protocole de diffusion. Nous couvrons ainsi les grands modèles de communication, en offrant une base adéquate de construction de protocoles. Chaque protocole définit des objets de communication et un ensemble d'opérations.

Le protocole point à point synchrone

Le protocole de communication entre processus, défini dans le langage occam, en est un représentant typique. L'objet de communication est le canal. Ce protocole permet à un couple de *threads* d'échanger des messages de taille quelconque de façon synchrone. Ce protocole est traité en détail dans le reste de ce chapitre.

Le protocole d'accès direct à la mémoire

Il est conçu pour réaliser les protocoles de bas niveau nécessaires au système d'exploitation et aux paradigmes d'appel ou d'activation de processus à distance. Il permet de copier des portions de mémoire de n'importe quel processeur vers n'importe quel autre du réseau. Un descripteur d'état d'avancement de la communication est disponible pour le récepteur. Le protocole est asynchrone, à la fin de la communication, l'émetteur peut associer l'activation d'un *thread* sur le processeur récepteur.

Ce protocole est utilisé dans le chargement dynamique de données ou de *threads* et l'activation de ces derniers ; il a aussi des applications à la migration ou à l'implantation d'appel ou d'activation de procédures à distance.

Le protocole client-serveur

Il est conçu pour la communication entre plusieurs clients et un serveur recevant leurs requêtes. Les objets de communication sont des *ports*. Chaque *port* correspond à une file d'attente de messages débitant des messages suivant une politique premier entré, premier sorti. Plusieurs émetteurs peuvent utiliser simultanément le même port. De même, une *tâche* peut disposer de plusieurs *threads* recevant sur un *port*.

L'émission sur un *port* est non bloquante, si le *port* n'a plus de place dans sa file pour recevoir un message, le système prévient l'émetteur. La réception est bloquante si aucun message n'est disponible. Lors d'un appel de procédure à distance, l'appelant doit réaliser une émission sur le *port* serveur suivie d'une réception sur le *port* qu'il a déclaré comme étant son *port* de retour. Ce protocole est présenté dans [BiNe84], [BRT87].

Ces deux derniers protocoles sont examinés en détail dans [Langue91].

Le protocole de diffusion

La diffusion dans les réseaux à connectique irregulière et modifiable est un problème complexe. Ce protocole est cependant nécessaire notamment pour le contrôle de l'exécution de processus parallèles (voir section 3.5.2) et pour des actions propres au système, comme par exemple la recherche de la localisation d'un objet qui peut impliquer la diffusion d'une demande à un ensemble de processeurs. Il existe des propositions dans la littérature, par exemple [ChaMa84], [JoBi88] et [KTFB89]. Une modélisation a été aussi proposée par Tricot dans [Tricot84].

Deux modes de diffusion sont étudiés par Langué dans [Langue87] : les diffusions à contrainte forte et lâche. Dans le premier, l'émetteur désigne un ensemble de récepteurs qui doivent *tous* recevoir le message, ils se synchronisent fortement. Dans le second, l'émetteur désigne un ensemble de récepteurs qui peuvent recevoir le message *s'ils sont à l'écoute* de cette communication ; il y a alors synchronisation lâche. Il est possible de combiner les deux modes. L'implantation de ce protocole dans Parx est actuellement en cours.

8.4 Introduction au protocole Occam réparti

Occam est un langage de haut niveau basé sur le modèle de processus séquentiels communicants (CSP) d'Hoare [Hoare78]. Les éléments de base du langage sont des processus ne partageant pas de variables et qui échangent des messages par l'intermédiaire de canaux point à point unidirectionnels. Des constructeurs du langage permettent de combiner des processus de manière séquentielle ou parallèle. Nous n'allons examiner ici que les caractéristiques concernant les mécanismes de communication, le lecteur intéressé davantage par le langage occam en trouvera la description complète dans [Occam2].

Une des caractéristiques notables du langage occam est le protocole de communication auquel nous nous intéressons ici. Son utilisation a été cependant limitée par la réalisation pratique des compilateurs. Ces derniers ne permettent pas la communication par rendez-vous entre deux processus s'exécutant sur des transputers qui ne sont pas voisins. Nous considérons dans le reste de ce chapitre la réalisation du protocole occam étendu à tout le réseau.

8.4.1 Communication et synchronisation dans occam

L'échange de messages entre deux processus suit un protocole synchrone et bloquant, que nous appellerons indifféremment *rendez-vous* ou *protocole occam*. Ce protocole permet à la fois la synchronisation et l'échange de données entre deux processus, que nous appellerons respectivement E (pour Emetteur) et R (pour Récepteur), par l'intermédiaire d'un objet de communication appelé canal. Un canal c est un objet privé et non cessible qui, de sa création à sa destruction, permet la communication entre un couple (E, R) donné. Le processus E ne peut qu'émettre sur c par l'instruction `Out` : c ! *expression*, tandis que R ne peut que recevoir de c par une instruction `In` : c ? *variable*. L'effet global est assimilable à une affectation du type : *variable* = *expression*.

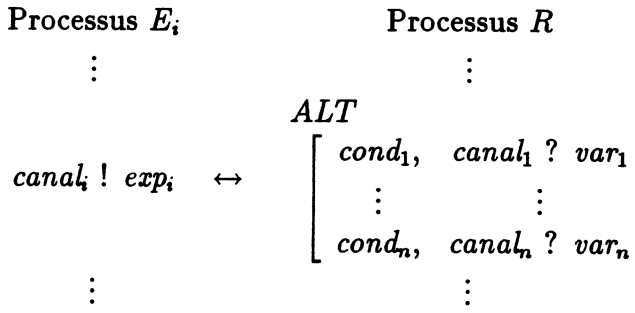
Le protocole est appelé *rendez-vous* car le transfert d'un message ne peut avoir lieu que si les deux processus participants à la communication sont prêts, E pour émettre et R pour recevoir. Le *rendez-vous* est possible dans deux situations que nous examinons ci-dessous.

Processus E		Processus R
⋮		⋮
canal ! exp	$\leftrightarrow$	*canal ? var*
⋮		⋮

rendez-vous inconditionnel : Les deux processus atteignent indépendamment l'un de l'autre le point de communication. Le premier arrivé attend l'autre, et lorsque le deuxième arrive, les processus échangent le message s'il y en a un[2]. Ils continuent ensuite d'exécuter le reste de leur code respectif. Il est nécessaire de remarquer que si l'un des processus n'arrive jamais au rendez-vous, l'autre restera bloqué pour toujours.

rendez-vous conditionnel : l'alternative. Elle introduit le non-déterminisme pouvant avoir lieu lors de communications multiples entre un récepteur et plusieurs émetteurs. Dans l'alternative, la suite du code du processus R est un ensemble de processus gardés par une action `In` sur un canal. Le processus R est donc en attente simultanée sur plusieurs canaux, le *rendez-vous* ne peut avoir lieu que sur un seul canal et un seul, choisi par R.

Une garde est composée d'une *condition* en forme d'expression booléenne et d'une action `In`[3]. Lorsque le processus R arrive à l'alternative, il évalue les conditions et choisit de communiquer sur l'un des canaux dont la condition est vraie et le processus correspondant est présent sur le canal. Si plusieurs correspondants sont au *rendez-vous*, R choisit un d'une façon non déterministe. Si aucun correspondant n'est présent, il s'endort ; il sera réveillé par le premier processus qui arrivera au *rendez-vous* sur un canal dont la condition est vraie. R ne communique donc qu'avec un et un seul des correspondants possibles.



8.4.2 Limitations du transputer et du compilateur Occam

Aussi bien le transputer que le compilateur occam, tout au moins jusqu'aux versions aujourd'hui disponibles : le T800 et Occam2 respectivement, imposent

[2]En occam, le message de la taille la plus petite est un octet, type BYTE. En langage assembleur transputer par contre, on peut faire une synchronisation sans transférer vraiment des données.

[3]En occam les gardes ne peuvent contenir des `Out`s sur les canaux.

une limite sur la programmation vraiment parallèle :

> *la communication entre deux processus E et R s'exécutant sur deux processeurs qui ne sont pas voisins dans un réseau de transputers n'est pas directement prise en charge ni par le transputer ni par le compilateur.*

Le protocole occam n'est pas supporté sur un réseau de transputers. Cette constatation nous a conduit à proposer des solutions pour l'implantation de ce que nous appelons le *Protocole Occam Réparti*, visant l'obtention du vrai occam parallèle. Une première solution, conçue par l'auteur de cette thèse, a été présentée dans [Gonz88]. Il s'agit d'une implantation sur occam-TDS que nous reprendrons dans la section 9.3[4]. Le protocole occam réparti, écrit en langage C est maintenant incorporé comme le protocole de base de Parx.

Remarquons finalement que le transputer T9000 offrira des *canaux virtuels* directement implantés sous le contrôle du matériel ; le concept de canal virtuel est la base pour obtenir le protocole occam réparti, il n'est autre chose qu'un canal au sens occam étendu sur le réseau.

8.5 Définition de canal virtuel

Il existe deux types de canaux en occam : les *canaux logiques*, qui ne sont qu'une place mémoire à laquelle les processus accèdent pendant le déroulement de la communication, et les *canaux physiques*, représentés aussi par une adresse mémoire et qui réalisent le protocole de *rendez-vous* entre deux transputers voisins, à travers un lien physique, comme s'il s'agissait d'un *rendez-vous* purement local.

Le problème est que le nombre de canaux logiques est déterminé par l'utilisateur selon ses besoins et n'est limité que par la capacité de mémoire ; en contrepartie, le nombre de canaux physiques est limité par le nombre de liens, en l'occurrence, les quatre liens du transputer. Une application qui s'exécute sur un seul transputer peut avoir une quantité virtuellement illimitée de canaux. Ce n'est pas possible d'exécuter la même application sur un réseau de transputers où les canaux sont limités par les quatre liens du transputer et la connectique du réseau sans y rajouter du logiciel permettant le routage et le multiplexage des liens logiques sur les liens physiques.

[4]Les études visaient la modification du compilateur occam, qui aurait été la solution idéale, malheureusement, nous n'avons jamais eu l'accès aux sources. La même idée a inspirée M. Debbage, M. Hill et D. Nicole de l'université de Southampton, qui ont développé un logiciel appelé Virtual Channel Router (VCR), présenté in extenso dans [DHN90]. Il est à remarquer que ce logiciel a pu se réaliser avec le concours direct d'Inmos.

La virtualisation de la notion de canal vise l'élimination de ce problème, en offrant aux utilisateurs une quantité virtuellement illimitée de canaux étendus sur tout le réseau. Le canal ne peut plus être juste une place mémoire, le canal virtuel (réparti) a besoin de deux extrémités pouvant se placer n'importe où dans le réseau de processeurs.

Définition : Un *canal virtuel* est un canal point à point unidirectionnel, ayant une extrémité *émettrice* et une extrémité *réceptrice* pouvant se placer indépendamment l'une de l'autre.

Remarque : La notion de canal virtuel contient celle du canal, rien n'empêche que les deux extrémités soient confondues dans une seule et unique place mémoire.

L'extrémité émettrice, dénotée `canal_e` accepte une action du type `Out (!)` et l'extrémité réceptrice, dénotée `canal_r` accepte une action du type `In (?)`.

Le processus E qui émet sur le canal doit connaître l'extrémité distante du canal, c'est-à-dire `canal_r`, tandis que le processus R doit connaître l'extrémité `canal_e`. Dans le cas d'occam actuel, les deux extrémités sont confondues en une seule entité connue des deux processus.

Cette notion de connaissance implique le besoin de mécanismes de *désignation* assimilables à ceux nécessaires à la désignation des objets dans un système réparti [CORNA81, LeBe88]. Ce problème ne concerne pas seulement le protocole occam réparti, ni la notion de *canal virtuel* ; il concerne le système Parx comme un tout, par conséquent, nous faisons ici abstraction de le méthode de désignation des extrémités des canaux virtuels, nous supposons que le mécanisme adéquat a été défini par le système d'exploitation. Une thèse actuellement en cours dans l'équipe **SYMPA** devra définir les politiques de gestion des objets dans Parx.

Cette notion de canal virtuel est utilisée par le transputer T9000 pour produire exactement le même effet que nous proposons ici. Dans ce cas, les extrémités des canaux virtuels sont des structures de données logées en mémoire.

8.6 Le protocole occam réparti : problèmes à résoudre

Outre le fait que le protocole occam réparti doive maintenir la sémantique du protocole "local" décrit dans la section 8.4.1, il doit répondre aux critères suivants :

- minimisation des communications, donc des messages "de contrôle" à échanger entre l'émetteur et le récepteur pour assurer la sémantique du protocole,
- prise en compte des cas d'alternatives réparties,
- possibilité d'émettre sur un canal des messages de taille variable,
- être sans interblocage.

Deux problèmes se posent maintenant que le canal n'est plus une seule place mémoire et que le message doit être transferé à travers le réseau. Le premier s'énonce ainsi : *où se produit le* rendez-vous *?, dans quelle extrémité du canal virtuel ?*, la réponse conduit à la définition des actions du protocole. Le second est celui relatif aux contraintes dues à la taille des messages acceptés par le réseau, nous savons que la taille est fixe et bornée par la taille des paquets, le mécanisme de *découper-réassembler* doit être mis en place avec la définition d'un mécanisme de contrôle de flux approprié.

Nous examinons ces deux problèmes dans les sections suivantes.

8.7 Alternatives de placement du point de *rendez-vous*

Pour mieux comprendre cette section, le lecteur ne connaissant pas bien l'implantation transputer du protocole occam peut se rapporter à l'annexe B. Pour réaliser le protocole occam réparti, nous allons considérer les trois possibilités suivantes :

- placer le point de *rendez-vous* dans l'extrémité `canal_r` du canal, c'est-à-dire du côté du récepteur,
- placer le point de *rendez-vous* dans l'extrémité `canal_e` du canal, c'est-à-dire du côté de l'émetteur,
- placer le point de *rendez-vous* dans l'extrémité `canal_r` du canal, et placer une zone tampon du côté récepteur.

Examinons maintenant les trois alternatives plus en détail en étudiant leurs avantages et inconvénients comparatifs. Les paramètres d'évaluation sont : la quantité de mémoire requise par le protocole et son efficacité en temps d'établissement du *rendez-vous*, mesure indiquée essentiellement par la quantité de paquets de contrôle échangés pendant le déroulement de la communication.

Par la suite, il faut toujours se rappeler le comportement générique des protocoles, du fait qu'il y a une instance d'interprétation sur chaque extrémité du canal virtuel et que la structure de l'interpréteur est formée d'un serveur, un récepteur et un émetteur.

Soit :

R_r, S_r **et** E_r le récepteur, serveur et émetteur du protocole du côté **`canal_r`** respectivement.

R_e, S_e **et** E_e le récepteur, serveur et émetteur du protocole du côté **`canal_e`** respectivement.

8.7.1 *rendez-vous* dans l'extrémité `canal_r`

rendez-vous inconditionnel	*rendez-vous* en alternative
$E \rightarrow R$ émetteur prêt $E \leftarrow R$ récepteur prêt $E \Rightarrow R$ envoi du message	$E_1 \rightarrow R$ émetteur 1 prêt ⋮ ⋮ $E_n \rightarrow R$ émetteur n prêt $E_i \leftarrow R$ { récepteur prêt, émetteur i selectionné $E_i \Rightarrow R$ envoi du message

Figure 8.2: *Rendez-vous* dans l'extrémité réceptrice.

La séquence de requêtes et transfert de données est illustrée dans la figure 8.2[5].

rendez-vous inconditionnel : l'action `Out`

Lorsque E est au *rendez-vous*, S_e envoie une requête de **`demande_de_rdv`**, adressée au **`canal_r`**, il prévient R_e de cette demande[6]. R_e se met alors en attente d'une

[5]Les messages de contrôle sont notés $\leftarrow\rightarrow$, les messages de donnés $\Rightarrow$.

[6]Lors de la mise en œuvre, S_e peut passer au R_e l'adresse et la longueur du message, celui-ci utilisera cette information pour trouver le message quand il faudra l'envoyer.

requête de **rdv_accepté**, laquelle déclenche l'envoi du message. A la fin du transfert du message, R_e réveille le processus utilisateur qui a demandé la communication.

Lorsque la requête de **demande_de_rdv** est reçue par R_r, il doit d'abord examiner l'état de l'extrémité **canal_r** puis réaliser l'une des deux actions suivantes :

1. Si R est déjà au *rendez-vous*, il faut envoyer une requête de **rdv_accepté** vers l'extrémité **canal_e**.

2. Si R n'est pas au *rendez-vous*, il faut mettre la requête dans l'extrémité **canal_r** en attente de l'arrivée de R.

rendez-vous inconditionnel : l'action In

Lorsque R arrive au *rendez-vous*, S_r ne fait que prévenir R_r, lequel doit examiner l'état de l'extrémité **canal_r** et réaliser l'une des deux actions suivantes :

1. Si E est au *rendez-vous*, R_r envoi une requête de **rdv_accepté** et se prépare à recevoir le message. Lors de la réception du message, les données sont placées directement dans le tampon réservé par l'utilisateur. A la fin de la réception, le processus utilisateur, jusqu'alors endormi, est réveillé par R_r.

2. Sinon, R_r marque la présence de R au *rendez-vous* et se met en attente d'une requête de **demande_de_rdv**.

La sémantique du protocole est maintenue, on a dû cependant introduire une aymétrie entre les actions **Out** et **In**. Le protocole est sans interblocage car il n'existe pas de dépendance cyclique entre les paquets qui contiennent le même type de requête.

rendez-vous en alternative

Le *rendez-vous* conditionnel est bâti aisément car le protocole de base a déjà introduit l'assymétrie nécessaire : la décision finale du *rendez-vous*, concrétisée par une requête de **rdv_accepté** et une seule, est prise dans l'extrémité **canal_r**.

La quantité de paquets de contrôle est bornée par $(n + 1)$, où n est le nombre de canaux impliqués dans l'alternative. Par contre, il faut deux messages de contrôle pour établir un *rendez-vous* inconditionnel.

Aucun espace mémoire additionnel n'est nécessaire.

rendez-vous inconditionnel	*rendez-vous* en alternative
$E \leftarrow R$ récepteur prêt $E \Rightarrow R$ envoi du message	$E_1 \leftarrow R$ récepteur 1 prêt ⋮ ⋮ $E_n \leftarrow R$ récepteur n prêt $E_1 \rightarrow R$ émetteur 1 prêt ⋮ ⋮ $E_n \rightarrow R$ émetteur n prêt $E_i \leftarrow R$ { récepteur prêt, émetteur i selectionné $E_i \Rightarrow R$ envoi du message

Figure 8.3: *Rendez-vous* dans l'extrémité émettrice.

8.7.2 *rendez-vous* dans l'extrémité `canal_e`

La séquence de requêtes et transfert de données est illustrée dans la figure 8.3.

rendez-vous inconditionnel : l'action `Out`

Lorsque E arrive au *rendez-vous*, S_e ne fait que prévenir R_e, lequel doit examiner l'état de l'extrémité **`canal_e`** et réaliser l'une des deux actions suivantes :

1. Si R est au *rendez-vous*, R_e commence le transfert des données. A la fin du transfert, le processus utilisateur, jusqu'alors endormi, est réveillé par R_e.
2. Sinon, R_e marque la présence de E au *rendez-vous* et se met en attente d'une requête de **`demande_de_rdv`**.

Dans cette solution il n'y a pas de requête de **`rdv_accepté`**.

rendez-vous inconditionnel : l'action `In`

Lorsque R arrive au *rendez-vous*, S_r ne fait que prévenir R_r, lequel envoie tout de suite une requête de **`demande_de_rdv`** et se met en attente du message. La sémantique du protocole est maintenue, il est sans interblocage pour la même raison que la première solution. Un seul message de contrôle est nécessaire au *rendez-vous* inconditionnel.

rendez-vous en alternative

Le placement du point de *rendez-vous* du côté de l'émetteur est très efficace pour le *rendez-vous* inconditionnel, par contre, la complexité du *rendez-vous* en alternative est considérablement augmentée ; il n'est pas difficile de comprendre

pourquoi : dans une alternative c'est toujours R qui prend la décision du *rendez-vous*, or si du côté de l'émetteur on n'envoie pas de requête, lorsque R arrive au *rendez-vous*, S_r est obligé de diffuser une requête de **demande_de_rdv** à toutes les extrémités **canal_e** participant dans l'alternative. L'arrivée d'une telle requête à une extrémité émettrice déjà au *rendez-vous* déclenche l'envoi d'une requête de **rdv_accepté** ; il peut y avoir jusqu'à n requêtes de ce type, parmi lesquelles une et une seule sera prise en compte, d'où l'inefficacité.

La quantité de paquets de contrôle n'est alors bornée que par $(2n + 1)$, mais le principal problème de cette solution est le contrôle correct de la suite du programme. Dans occam, si le processus R est une boucle, l'entrée successive dans l'alternative peut accepter des *rendez-vous* avec des E_i étant arrivés au *rendez-vous* et qui n'ont pas été pris en compte dans la boucle précédente, le protocole réparti doit maintenir ce fonctionnement ; par ce fait, l'instance du protocole doit avoir une durée égale à celle du programme et le comportement de R_r doit être obligatoirement celui de la première solution.

8.7.3 *rendez-vous* dans l'extrémité canal_r avec tampon

rendez-vous inconditionnel	*rendez-vous* en alternative
$E \Rightarrow R$ { émetteur prêt + qques données	$E_1 \Rightarrow R$ { émetteur 1 prêt + qques données
$E \leftarrow R$ récepteur prêt	⋮ ⋮
$E \Rightarrow^* R$ reste du message	$E_n \Rightarrow R$ { émetteur n prêt + qques données
	$E_i \leftarrow R$ { récepteur prêt, émetteur i selectionné
	$E_i \Rightarrow^* R$ reste du message

Figure 8.4: *Rendez-vous* dans l'extrémité réceptrice et zone tampon.

La deuxième solution n'est donc pas envisageable car la gestion des alternatives nous fait revenir à la première. L'inconvénient de celle-ci est qu'elle a besoin de deux paquets de contrôle avant de commencer le transfert du message. Cette troisième méthode n'est qu'une modification de la première dans laquelle, lorsque E arrive au *rendez-vous*, la requête de **demande_de_rdv** est accompagnée de quelques données qui sont stockées du côté récepteur dans un tampon temporaire géré par le protocole. Lorsque le message complet peut être placé dans le

tampon, la requête de **rdv_accepté** envoyée par R_r met fin au *rendez-vous* et R_e peut réveiller tout de suite le processus E ; un seul paquet de contrôle est alors nécessaire pour le *rendez-vous* inconditionnel et pour l'alternative, le protocole est illustré dans la figure 8.4.

Cette méthode nécessite un tampon pour chaque extrémité réceptrice de chaque canal virtuel, l'encombrement mémoire est donc beaucoup plus important ; un dimensionnement correct de la taille du tampon, qui n'est pas nécessairement la taille des paquets, peut permettre de réaliser tous les *rendez-vous* avec un seul paquet de contrôle dans beaucoup d'applications. Nous pensons particulièrement aux applications de calcul intensif où ne sont échangés que des réels, on peut dimensioner la taille de la zone tampon additionnelle à la taille d'un réel. Dans ce cas, il est sûr qu'un seul paquet peut contenir tout le message, le coût en mémoire additionnelle est alors minimun et le gain en temps d'établissement du *rendez-vous* réparti est considérable.

8.8 Le mécanisme de *découper-réassembler*

Le découpage de gros messages en morceaux de taille plus petite, et l'envoi du message morceau par morceau, est nécessaire pour le transfert de messages de taille variable sur le réseau de routage qui ne transfère que des paquets de taille fixe.

La solution à ce problème est bien connue et facilitée par la sémantique du protocole occam réparti : au moment du *rendez-vous*, le récepteur a réservé la place mémoire pour stocker tout le message, de cette manière, le réassemblage se fait en remplissant cet espace au fur et à mesure de l'arrivée des paquets. La reconstruction correcte du message requiert cependant quelques soins. Par exemple, il n'est pas simple de reconstruire correctement un message, sans recourir à des procédés compliqués et à l'utilisation de tampons additionnels, sous la combinaison des trois conditions suivantes (nous supposons que les morceaux sont numérotés en forme consécutive lors du découpage du message et le numéro est inclus dans l'*en-tête* du paquet) :

i) le message est découpé en morceaux de taille différente. Les paquets transfèrent donc des morceaux de longueur différente, la longueur utile étant indiquée dans l'en-tête de chaque paquet,

ii) la longueur de la fenêtre régulant le flux est $\Omega > 1$,

iii) le routage ne préserve pas l'ordre des paquets, c'est-à-dire que les paquets peuvent suivre des routes différentes et arriver à destination sans préserver

l'ordre avec lequel ils furent émis.

Il suffit néanmoins d'éliminer l'une quelconque de ces trois conditions pour faciliter considérablement la reconstruction du message :

élimination de i) le message est découpé en morceaux de la même longueur ; même s'ils arrivent en désordre et par groupes, il suffit de remplir le tampon de réception en plaçant les morceaux à l'emplacement indiqué par son numéro.

élimination de ii) si $\Omega = 1$, les morceaux sont envoyés l'un après l'autre et acquittés individuellement, il suffit alors de remplir l'espace mémoire du récepteur au fur et à mesure de l'arrivée des paquets. La longueur des morceaux n'est donc pas importante et il est impossible pour le routage de ne pas préserver l'ordre, il n'y a jamais deux morceaux du même message en transit simultanément.

élimination du iii) si le routage assure que les paquets arrivent dans le même ordre qu'ils ont été émis, on revient à la situation décrite ci-dessus, même si $\Omega > 1$, les paquets arriveront dans l'ordre d'émission.

Le choix consistant à éliminer une de ces conditions, qui entravent le réassemblage des messages, dépend des considérations d'efficacité. Nous pensons que la meilleure solution consiste à éliminer i) car ii) et iii) imposent des contraintes sur le routage et le contrôle de flux, ce qui n'est pas souhaitable.

Remarquons néanmoins que le contrôle de flux proposé par Inmos pour la paire T9000-C104 consiste à fixer $\Omega = 1$, ce qui à notre avis diminuera notablement l'efficacité d'utilisation de la bande passante du réseau.

8.9 Conclusions

Nous n'avons examiné dans ce chapitre les principaux aspects conceptuels concernant la construction de protocoles de communication pour Parx, dont il faut souligner notamment la structure générique qui permet, à n'importe quel protocole, d'offrir la même interface, aussi bien vis-à-vis de l'utilisateur que vis-à-vis du réseau de routage. D'autres protocoles peuvent donc venir se rajouter aisément, comme c'est le cas du protocole réparti de gestion de fautes de pages [Castro91].

Nous avons examiné de plus près la réalisation du protocole occam réparti, l'un des protocoles de base, qui permet à Parx d'offrir le support nécesaire au modèle de programmation par processus communicants.

Partie IV

Mise en œuvre

Cette partie est composée d'un seul chapitre dans lequel nous voulons montrer quelques aspects relatifs à la mise en œuvre du noyau de communication. Nous mettons l'accent sur la mise en œuvre du protocole de rendez-vous *réparti. Notre approche consiste à faire une présentation allégée, en évitant d'entrer dans les détails techniques de la programmation.*

Chapitre 9

Aspects de mise en œuvre et mesure de performances

9.1 Introduction

Depuis 1986 environ, plusieurs développements pratiques accompagnent les études théoriques réalisées dans notre équipe de travail autour des architectures, langages, environnements de programmation et systèmes massivement parallèles.

Dans ce chapitre, nous faisons un bref bilan des aspects relatifs à la mise en œuvre du noyau de système que nous avons examiné, d'un point de vue plutôt théorique, tout au long de cet ouvrage. Ce noyau n'est qu'une partie d'un projet ambitieux dont les réalisations pratiques, au début menées en parallèle et indépendamment, se rassemblent maintenant autour de Parx, que nous proclamons être l'un des systèmes parallèles, en état de prototype, au sommet de l'art dans son genre.

Historique

Les premiers développements pratiques étaient fortement influencés par le transputer, le langage occam et l'architecture massivement parallèle et reconfigurable : le Supernode. Nous avons précisé dans la section 8.4 certaines limitations du compilateur occam qui le rendent peu adapté à la programmation directe des réseaux de transputers, une chaîne de développement des programmes occam fut alors nécessaire [MEMT88, EMM89, Eudes90].

Simultanément, afin de faciliter les expériences sur les transputers, on développa un programme assembleur permettant de programmer les transputers à bas niveau et indépendamment du langage occam.

Le besoin d'un noyau de communication se faisait ressentir par tous ceux qui voulaient développer des applications pour des réseaux de transputers. D'un côté étaient ceux écrivant des programmes en langage occam sous le système TDS. D'un autre côté étaient ceux qui avaient besoin de fonctionnalités qu'occam n'était pas en mesure de fournir[1]. Pour les premiers, et par un travail personnel de l'auteur de cette thèse, un noyau de communication offrant des canaux virtuels et le protocole occam réparti, discuté dans le chapitre 8, fut écrit en occam sous TDS et mis à la disposition de la communauté sous la forme d'une librairie, ce travail est au cœur de notre discussion dans les sections suivantes de ce chapitre.

Pour les seconds, un travail simultané, mené par plusieurs membres de l'équipe SYMPA, aboutit à un noyau de communication écrit essentiellement en assembleur et langage C. Ce noyau fut mis en œuvre aussi sur des machines UNIX, telles que SUN et APOLLO, sur une couche de simulation du transputer qui permet la réalisation de réseaux mixtes, comportant de transputers et des machines UNIX ; l'importance de cette démarche est que l'on pouvait alors observer le comportement du noyau et la tâche de déverminage des programmes fut énormément facilitée. Le noyau a été ainsi validé.

Le cadre se complète par d'autres travaux relatifs au contrôle de la reconfigurabilité du Supernode, c'est sans doute le sujet le plus difficile, compliqué aussi bien par sa complexité intrinsèque que par le manque d'expérience. Nous y avons consacré plusieurs sections dans cette ouvrage, en apportant notre contribution à ce domaine. La reconfiguration a été étudiée aussi d'un point de vue théorique par Sakho et Mugwaneza [MuSa91] et d'un point de vue pratique par Waille [Waille91].

Dans ce chapitre nous allons examiner les aspects remarquables des réalisations pratiques. Nous examinons d'abord quelques aspects concernant la mise en œuvre de la couche de routage des messages. Ensuite nous verrons l'implantation du protocole occam réparti, pour finir avec quelques résultats concernant les mesures de performance.

9.2 Routage des messages : mise en œuvre

Les structures de données seront décrites en utilisant un format proche du langage C, elles sont cependant générales et peuvent être appliquées au langage occam

[1]Rappelons en particulier qu'occam n'offre pas de manipulation de pointeurs et pas d'allocation dynamique de la mémoire.

par exemple en adaptant convenablement le code.

L'objet manipulé

L'information de base circulant sur le réseau est le paquet ; il est constitué d'un *en-tête* et d'un *corps*. Ce dernier contient les données transférées par le paquet et ne correspond qu'à une suite d'octets sans aucune structure du point de vue du routage. L'*en-tête* contient l'information nécessaire à l'acheminement du paquet à travers le réseau et à son traitement lorsqu'il arrive à destination. La structure suivante est générique, elle doit s'adapter lors d'une implantation particulière[2] :

```
typedef  t_proc_id    int;  /* identificateur de processeur */
typedef  t_chan_id    int;  /* identificateur de canal */
typedef  t_prot_id    int;  /* identificateur de protocole */
typedef  t_prot_arg   *int;  /* liste d'arguments du protocole */
typedef  t_trait_id   int;  /* identificateur de traitement */

typedef  struct  t_paquet_header
{
   t_proc_id   proc_dest;
   t_proc_id   proc_scr;
   t_prot_arg  arguments;
   t_prot_id   protocol;
}  p_header;

typedef struct  t_paquet
{
   p_header  header;
   t_byte    *body;
}  paquet;
```

La longueur d'un paquet est figée par le noyau pour toute communication, indépendamment du protocole. La liste d'arguments peut varier d'un protocole à un autre, la taille du *corps* doit s'ajuster en conséquence, elle est cependant fixe pour un protocole donné.

Couramment, le rapport entre la taille de l'*en-tête* et la taille du *corps* est une mesure d'efficacité qui a une influence sur le choix de la taille du paquet. En effet,

[2]Nous utilisons un type **int** qui doit être ajusté en **short** ou **long**, etc. selon les besoins spécifiques de l'implantation.

ce n'est que le *corps* qui contient les données utiles à l'utilisateur. La frontière entre le *corps* et l'*en-tête* est a discuter, rien n'empêche de placer l'identification du protocole et la liste des arguments comme faisant partie du *corps*. Pour nous, toute information devant se répeter pour des données différentes ; autrement dit, toute information nécessaire au contrôle de la communication, fait partie de l'*en-tête*, d'où la définition de la structure donnée ci-dessus.

A propos de l'efficacité du routage

Dans la section 6.4 nous avons dit que la fonction R était efficace, en ce qui concerne le temps, car la détermination du lien de sortie ne requiert qu'un accès mémoire, ce n'est cependant que l'efficacité du routeur. En réalité, l'efficacité du routage dépend du temps passé entre le moment où un paquet arrive au processeur et le moment où il est réémis sur un lien de sortie. Ce temps est la somme du temps de traitement de l'ensemble *récepteur-routeur-émetteur*. La plupart du temps, on mesure ceci sans considérer le temps passé dans une file d'attente, celui-ci est variable et dépend de la charge du réseau.

La minimisation du temps de routage est très importante pour minimiser le temps global de transfert d'un message. Les choix de mise en œuvre des divers processus du routage et des interactions de l'ensemble sont donc très importants pour obtenir un routage efficace. Les fonctions *récepteur* et *routeur* peuvent être combinées de manières différentes ; examinons deux possibilités.

solution 1: Pour chaque lien d'entrée, les fonctions *récepteur* et *routeur* sont réalisées par un seul processus ; la décision de routage serait prise dès la réception du paquet. La figure 9.1a est une illustration de cette solution.

solution 2: Le *routeur* est réalisé indépendamment du *récepteur*, les figures 6.1 et 9.1b représentent cette situation ; dans le premier cas un seul processus réalise le routage de tous les paquets arrivant sur le processeur. Dans le second cas un routeur est associé à chaque lien d'entrée.

Le grand inconvénient de la solution 1 est que le *récepteur* sera bloqué si le *routeur* se bloque sur une file d'attente pleine : supposons qu'un paquet pq_1 arrive à un processeur par son lien l_0 et la fonction de routage l'envoie sur le lien de sortie l_1, si la file de l_1 est pleine, le *routeur* sera bloqué et par conséquant le *récepteur* aussi. La réception de tout autre paquet pq_j, arrivant par l_0, est arrêtée même si la route suivie par pq_j utiliserait un lien de sortie $l_j, j \neq 1$ pour lequel la file n'est pas pleine. On introduit donc une interférence entre les routes qui est évitée dans la solution 2 par la séparation des fonctions *réception* et *routeur* en des processus distincts.

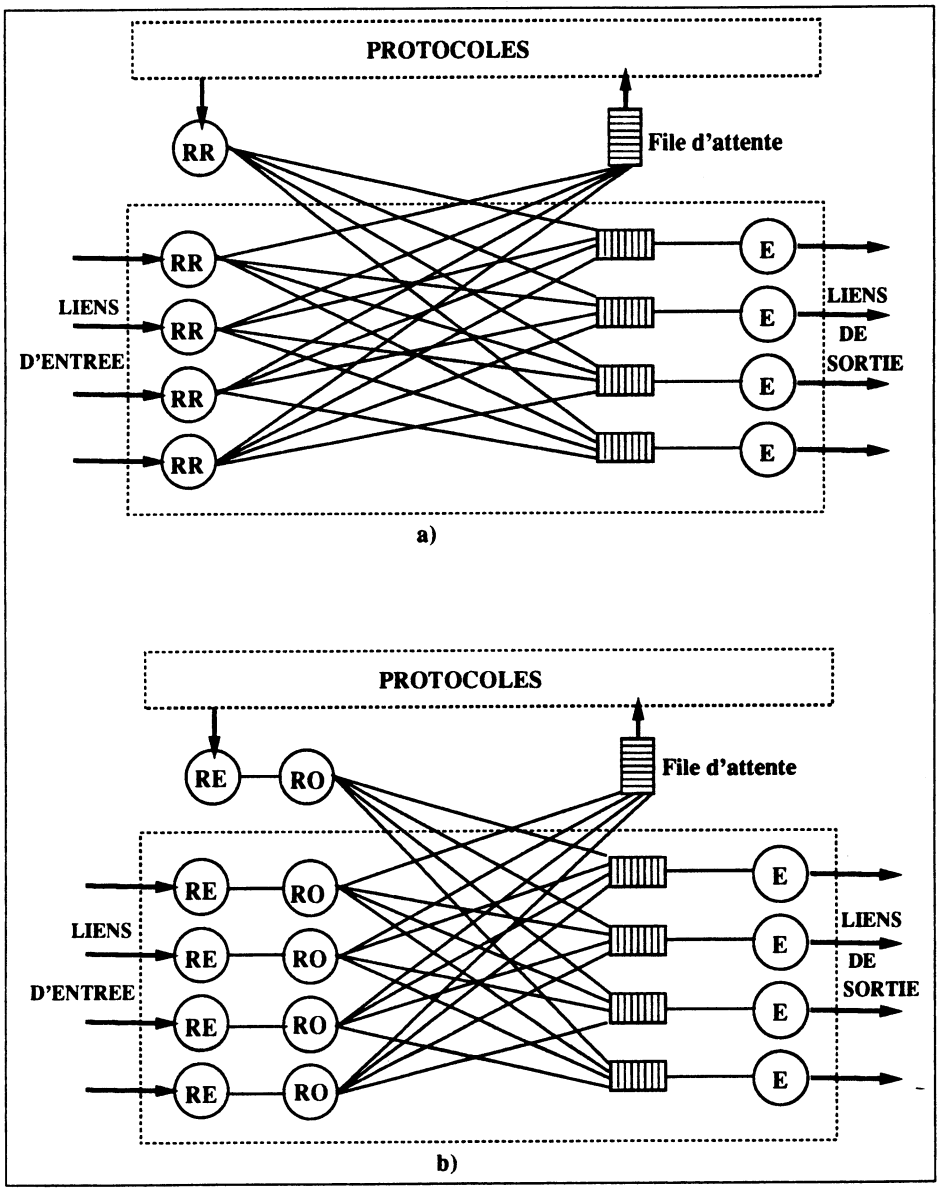


Figure 9.1: Alternatives de mise en œuvre du niveau routage. **RR** : *Récepteur-Routeur*, **RE** : *Récepteur*, **RO** : *Routeur*, **E** : *Emetteur*

Aucune solution ne peut cependant éviter le blocage temporaire d'un ensemble de paquets remplissant les files d'une seule route surchargée, ou d'un ensemble de routes qui ont des files communes. La solution à ce problème répose sur le contrôle de flux que nous avons examiné dans la section 6.6. Rappelons que le routage sans interblocage ne requiert que l'utilisation d'un tampon, de la taille d'un paquet, par lien (il n'y a pas besoin des files). Le maintien des files de longueur supérieure à un est nécessaire pour diminuer les possibilités de congestion et maintenir un bon débit de messages.

L'*émetteur* est séparé du *routeur* pour éviter que celui-ci puisse se bloquer sur la réémission d'un paquet sur un lien occupé, alors qu'il pourrait servir d'autres paquets utilisant des liens libres. L'utilisation des files indépendantes induit aussi la séparation de ces deux processus.

Nous considérons que la file d'attente de chaque *émetteur* est inclue dans celui-ci, cependant, si nous avons besoin de les différencier, nous parlerons de processus *file*. Les processus *récepteur*, *routeur*, *émetteur* et *file* sont tous constitués d'une bloucle se répétant pour toujours, sans autre condition que l'existence d'un paquet à traiter. Dans chaque boucle un paquet et un seul est traité.

Commentaires sur l'implantation en occam

Nous avons réalisé une mise œuvre en utilisant le langage occam. Il s'agit d'un noyau de communication qui réalise la virtualisation du protocole du *rendez-vous*, comme nous l'avons discuté dans le chapitre 8, implanté sur une couche de routage sans interblocage, basé sur le modèle discuté dans le chapitre 6 et l'algorithme présenté dans le chapitre 7. Le noyau, couramment appelé **COMKER**, a été mis à disposition des partenaires du projet **ESPRIT SUPERNODE** et présenté dans [GLM90].

Nous présentons ici nos conclusions sur l'utilisation du langage occam. Nous voulons simplement souligner ce que cette expérience nous a appris sur les problèmes d'efficacité d'implantation d'une couche de routage en langage occam pur[3]. Les paragraphes suivants caractérisent l'implantation.

1. Le transfert d'information entre les processus *récepteur, routeur et émetteur* doit se faire par communication par *rendez-vous* sur des canaux logiques. Un paquet doit être recopié sur des emplacements différents autant de fois qu'il est passé d'un processus à un autre. Ceci est dû aux faits suivants :

[3]Nous parlons de langage occam pur lorsqu'on n'utilise pas des incrustations en assembleur ou un autre langage de haut niveau.

- Le langage n'offre pas la manipulation de pointeurs qui permettraient de ne pas recopier le paquet en entier mais seulement son adresse.
- L'utilisation cohérente de variables partagées[4] n'est pas assurée lorsque les processus qu'y accédent s'exécutent en concurrence, comme c'est le cas de l'ensemble des processus de la couche de routage.

2. Dans les figures 9.1a) et b) on voit que chaque file d'attente des liens de sortie reçoit une entrée de chaque routeur. En occam, chaque ligne de la figure doit être un canal[5] et chaque processus *émetteur* doit donc mettre en œuvre un constructeur d'alternative, **ALT**, sur tous les canaux entrants. Or, la réalisation d'une alternative est coûteuse, car le code généré (voir annexe B) requiert l'ensemble des **EnableChans** et **DisableChans**, et cela chaque fois que le processus **émetteur** commence une nouvelle boucle et pour tous les canaux, même si certains d'entre eux ne sont jamais utilisés.

Utilisation du langage assembleur

Les deux problèmes soulignés ci-dessus peuvent être résolus par la programmation en langage assembleur du transputer, dans quel cas il n'est pas difficile de gérer des pointeurs ; cela permet de passer les paquets par référence en utilisant seulement leurs adresses, la gestion de files est simplifiée et on peut éviter d'utiliser l'alternative, le passage des paquets par des canaux internes est éliminé par une construction de listes circulaires.

Chaque file d'attente est réalisée par une liste circulaire, le *routeur* ne transfère pas le paquet vers une file mais le met en dernière position de la liste, l'*émetteur* ne fait que servir la liste correspondante en prenant tour à tour le paquet en tête de la liste. La structure (générique) des données d'une liste est la suivante :

[4]La philosophie du langage occam et du CSP est que les processus concurrents ne partagent pas de variables. Le partage de variables est accepté par le compilateur occam2 mais la cohérence des opérations concurentes sur ces variables n'est pas assurée.

[5]Rappelons qu'un canal occam est unidirectionnel et point à point.

```
typedef struct t_liste
{
  t_objet           *l_netx;
  t_paquet_header   *p_header;
  t_paquet_body     *p_body;
} liste;

typedef struct  t_link_sender
{
  t_proc_id   *sender_Wdesc;
  t_objet     *liste_first;
  t_objet     *liste_last;
} sender;
```

Nous savons qu'il existe un *émetteur* par lien. Une liste circulaire est servie par un et un seul *émetteur*, mais elle s'étend sur tous les *routeurs*. Lorsqu'une liste est vide, l'*émetteur* correspondant s'endort, il est réveillé par le *routeur* qui met un paquet dans la liste. Pour ce faire, le *routeur* a accès à l'espace de travail de l'*émetteur*, car il se trouve dans le champ **sender_Wdesc** de la structure **sender**.

Lorsqu'une liste est pleine, le routeur est temporairement bloqué en essayant de placer un paquet en queue de la liste. Cette situation ne devrait se produire que lors d'une congestion du réseau.

Les opérations sur les files, rajout d'un élément en queue et retrait d'un élément en tête de la file sont classiques en informatique et nous ne les montrons pas ici.

9.3 Mise en œuvre du protocole occam réparti

Nous décrivons ici la mise en œuvre en langage occam. Une réalisation en langage C, qui a été incorporé dans Parx et qui suit les mêmes principes, est décrite dans [GraFa90]. La meilleure solution pour la mise en œuvre des canaux virtuels et du protocole occam réparti aurait été d'incorporer les concepts, examinés dans le chapitre 8, directement dans le compilateur occam ; nous n'avons pas eu cette possibilité, faute d'accès au code source du compilateur. Il fallait donc réaliser un logiciel à la fois facile d'utiliser et compilable avec le compilateur occam normal. Le principal défi était alors de réaliser les canaux virtuels et le protocole occam réparti sans obliger l'utilisateur de ce logiciel à trop changer ses programmes originaux. Nous décrivons ci-dessous les aspects principaux de notre démarche.

9.3.1 Les canaux virtuels

Un canal virtuel ne doit pas être très différent de ce qu'est un canal normal occam entre deux transputers voisins. Un tel canal est construit en dessus d'un lien physique ayant une extrémité dans chaque transputer. L'établissement d'un canal occam entre deux transputers requiert les pas suivants :

pas 1 : Le lien physique est identifié par un numéro unique.

pas 2 : Dans chaque transputer il faut déclarer un canal en lui donnant un nom unique qui l'identifie. Du côté émetteur, on ne peut que réaliser une action `Out`, et du côté récepteur on ne peut que faire un `In`.

pas 3 : Les canaux déclarés dans le pas 2 doivent être "connectés" au lien physique.

Il est imposé qu'un lien physique ne peut être utilisé que par un canal dans chaque sens[6], et qu'un même canal ne peut être connecté qu'à un seul lien et un seul. Le programme occam correspond à la structure suivante :

```
 transputer t1:                          transputer t2:

-- associating logical to physical noms (pas 1)
VAL  link0out  IS  0:                   VAL  link0out  IS  0:
VAL  link0in   IS  4:                   VAL  link0in   IS  4:
-- more values                          -- more values

-- declarating channels (pas 2)
CHAN OF ANY  out:                       CHAN OF ANY  in:

-- PLACEing channels on physical links (pas 3)
PLACE  out  AT link0out:                PLACE  in  AT link0in:
-- more placements                      -- more placements

-- declaring processus which communicates through channels
PROC p1(CHAN OF ANY out)              PROC p2(CHAN OF ANY in)
   -- p1 code                             -- p2 code
  out ! expression                        in ? variable
   -- more p1 code                        -- more p2 code
```

L'établissement d'un canal virtuel doit suivre la même démarche mais adaptée aux nouveaux besoins, illustrons cela en établissant un canal virtuel entre deux

[6]Rappelons qu'un lien physique est bidirectionnel.

processus p_1 et p_2. Maintenant, ils peuvent s'exécuter sur des transputers non voisins. Supposons que p_1 s'exécute sur t_1 et que p_2 le fait sur t_2.

pas 1 : Un "lien virtuel" doit être établi entre les deux transputers. Ce lien virtuel, qui est réalisé par la couche de routage de messages, n'est rien d'autre qu'une route $route(t_1, t_2)$.

pas 2 : Une extrémité `canal_e` doit être déclarée dans le processeur t_1 et une extrémité `canal_r` doit être déclarée dans t_2. Chaque extrémité ainsi créée doit être connectée au lien virtuel.

Cette phase de "déclaration-connexion" doit être acceptée par le compilateur occam, il ne s'agit en fait que d'un mécanisme de *désignation* des objets dans un environnement réparti.

Chaque extrémité du canal virtuel sera désignée en utilisant un identificateur global unique, IGU, lequel contient deux champs, le premier correspond à un identificateur local de l'extrémité du canal virtuel, IL_e ou IL_r, et le second correspond à l'identificateur du processeur où se trouve l'extrémité.

$$IGU_e = \langle IL_e, t_1 \rangle$$

$$IGU_r = \langle IL_r, t_2 \rangle$$

L'IL permet de différencier plusieurs canaux virtuels construits sur la même route, dans la mise en œuvre il faut définir un tableau qui puisse contenir tous les identificateurs locaux nécessaires.

pas 3 : La "connexion" d'un canal virtuel à la $route(t_1, t_2)$ est faite tout simplement en incluant les IL_e et IL_r dans la liste d'arguments de l'*en-tête* des paquets échangés pendant le déroulement du protocole. L'*en-tête* doit donc contenir :

$\langle IGU_e, IGU_r \rangle$ pour les paquets allant de t_1 vers t_2.

$\langle IGU_r, IGU_e \rangle$ pour les paquets allant de t_2 vers t_1.

Leur adaptation à la forme générale d'un *en-tête*, définie dans la section 9.2 est aisée.

9.3.2 Structures des données échangées sur les canaux virtuels

Les canaux virtuels supportent un protocole variant qui permet aux processus d'échanger des données structurées comme celles définies par occam. Tous les types primitifs sont disponibles par l'utilisation du protocole occam suivant[7]

```
PROTOCOL msg
  CASE
    bool    ;  BOOL
    int     ;  INT
    int16   ;  INT16
    int32   ;  INT32
    int64   ;  INT64
    real32  ;  REAL32
    real64  ;  REAL64
    string  ;  INT::[]BYTE
:
```

Tous les canaux virtuels utilisent cette structure des données. Supposons que les identificateurs t_1 et t_2 soient connus de p_1 et p_2 respectivement.

Dans t_1, un tableau `[MAX.CHAN][MAX.TRA]CHAN OF msg out`, permet de déclarer `MAX.CHAN x MAX.TRA` extrémités émettrices. De même, dans t_2, un tableau `[MAX.CHAN][MAX.TRA]CHAN OF msg in`, permet de déclarer `MAX.CHAN x MAX.TRA` extrémités réceptrices. `MAX.CHAN` et `MAX.TRA` représentent le nombre maximum des canaux virtuels et de transputers respectivement[8]. Ces valeurs sont obligatoirement fixées par le noyau.

Supposons que les processus utilisent un $IL = 0$ pour communiquer, alors, le code nécessaire à une communication est :

```
      transputer t1                          transputer t2

out[0][t2] !  tag ; data                 in[0][t1] ?  CASE
                                           tag ; data--choisir le type
                                             ... action pour ce type
```

[7]Un PROTOCOL occam est une structure des données permettant d'échanger des données ayant des types différents sur un même canal.

[8]Cette déclaration des tableaux de taille maximale est obligatoire en occam faute d'allocation dynamique de mémoire.

Ce code est bien accepté par le compilateur occam2, ce qui était une des conditions de la réalisation du protocole occam réparti.

Pour faciliter la migration de processus, le noyau permet de placer les extrémités émettrice et réceptrice d'un même canal virtuel dans un même transputer, le code suivant est donc valide :

```
--Code on transputer t1:
PAR
  SEQ       --processus p1
    ... some Occam code
    out[chan.id][t1] ! tag ; data
    ... more Occam code
  SEQ       --processus p2
    ... some Occam code
    in[chan.id][t1] ?  CASE
      tag ; data
        ... action for this tag
    ... more Occam code
```

La communication locale passe par le noyau, elle prend plus de temps qu'une communication sur un canal occam normal, mais ceci permet de migrer les processus sans modifier le code. Un mécanisme de migration de processus pourrait déplacer le processus p_2 sur un transputer t_y, il suffit de remplacer t_1 par t_y dans le code du processus p_1 et la communication continue à être assurée par le noyau de communication de façon transparente au programmeur.

L'interface entre le programme utilisateur et le noyau

Lorsqu'un processus utilisateur réalise une communication en utilisant les primitives normales d'occam : le ! pour émettre et ? pour recevoir, il faut que le noyau soit capable de prendre en charge la communication et de réaliser le protocole occam réparti tel que nous l'avons défini.

Un mécanisme de très bas niveau, qui utilise le langage assembleur transputer, permet d'"attraper" les messages émis et de déclencher les activités nécessaires au protocole réparti. Ce mécanisme est très particulier, il utilise le fait que l'on peut réaliser une à une les instructions du constructeur **ALT** du langage occam[9].

[9]Voir l'annexe B.

Nous ne donnons ici qu'une description sommaire, les détails du mécanisme sont très techniques et relativement compliqués à expliquer in extenso.

Action du noyau dans l'extrémité émettrice

Dans le noyau, il existe un processus `trap(out)` qui est en haute priorité et dans l'état `attente d'alternative` sur le canal `out`. Ainsi, il fait croire au processus utilisateur que son correspondant sur le canal `out` est prêt[10]. Lorsque le processus utilisateur exécute l'action (`out ! tag;data`), il trouve le processus `trap` sur le canal, comme `trap` est en alternative, c'est lui qui prend la décision de communiquer, l'adresse du message est alors connue de `trap`. A partir de ce moment, le processus utilisateur reste endormi (il se croit en communication dans une alternative dans laquelle il pas encore été choisi), le processus `trap` réalise le protocole occam réparti avec son correspondant remote et ne réveille le processus utilisateur que lorsque le message a été transféré. Le processus `trap` est donc le *serveur* du protocole.

Dans ce cas particulier, la méthode de découpage-réassemblage est appliquée exclusivement aux messages du type `string`, qui sont les seuls à longueur variable.

Action du noyau dans l'extrémité réceptrice

C'est le processus `trap` qui déclenche les actions du protocole en envoyant une requête de `demande_de_rdv` vers le transputer où se trouve le récepteur. Rappelons que ce sont les indices des tableaux des extrémités des canaux virtuels qui permettent d'identifier les correspondants, ils sont donc bien connus de `trap`.

Du côté du récepteur, le *récepteur* du protocole est un processus `tampon(in)` qui reçoit les données du réseau et les envoie par le canal `in` qui correspond. Le processus `tampon` permet de libérer le noyau qui peut servir d'autres communications. Lorsque le processus utilisateur exécute (`in ? ....`) ; il se met en communication avec `tampon`, à la fin de la communication, celui-ci déclenche l'envoi d'une requête de `rdv_accepté` qui permettra au processus `trap` de réveiller le processus utilisateur émetteur.

Tout ceci est un peu plus compliqué lorsqu'il s'agit d'un message du type `string`, le découpage est réalisé par `trap`, et le réassemblage est réalisé par `tampon`, les paquets sont acquittés un à un et les processus utilisateurs ne sont réveillés que lorsque tout le message a été complètement transféré.

[10] Nous laissons de côté l'emploi des tableaux pour simplifier la notation.

Soulignons finalement que lorsque **tampon** réassemble le message, il le fait directement dans la place mémoire du processus récepteur ; il connaît l'adresse où le processus utilisateur veut recevoir le message.

Alternative occam étendue aux réseaux de transputers

La mise en œuvre du protocole occam réparti permet la réalisation du constructeur **ALT** du langage occam étendue sur tout le réseau de transputers. Suppossons que le processus p_5 s'exécutant sur un transputer t_5 doive réaliser des actions déterminées selon la réception de messages qui lui sont envoyés par des processus s'exécutant sur des transputers t_1, t_2, t_3, t_4 *et* t_5, dans ce dernier cas on a affaire à un message local au transputer t_5. Le code est le suivant :

```
ALT  i = 1  FOR  4
  in[chan.id][ti] ?  CASE
    tag ; data
      ... action associated to this tag
```

Les processus émetteurs auront un code comme ceci :

```
... user process action
out[chan.id][t5]  !  tag ; data
... more actions to be executed only if this channel
    was choosen by the ALT constructor in t5
```

Ce constructeur est donc une vraie alternative répartie, qui maintient la sémantique du **ALT** occam. Soulignons que la construction utilisée dans l'exemple est un cas particulier dans lequel l'identificateur du canal est le même pour chaque processeur, rien n'empêche aux processus émetteurs d'utiliser un identificateur de canal différent des autres, ou qu'un seul processus émetteur puisse avoir plusieurs canaux différents participant au même **ALT** réparti.

9.4 Mesures de performance

L'état du développement des travaux pratiques nous conduisent à la réalisation d'une étude sur les mesures de performances du noyau de communication. Cette étude est nécessaire pour mieux comprendre le comportement du noyau, ce que permettra un meilleur choix d'implantation aussi bien que l'optimisation du code.

Bien que tout au long de ce chapitre, dans le but de privilégier la clarté de l'exposé, nous avons préféré de donner une description de la réalisation en occam, l'étude des performances est réalisée sur le noyau écrit en langage C, lequel est d'ailleurs à la base de Parx.

Les résultats de notre étude sont présentés en détail dans l'annexe C. Ci-dessous nous ne présentons qu'un résumé.

Rappelons que notre objet d'expérimentation est l'architecture Supernode ; qui correspond, pour simplifier, à un réseau de transputers. Nous sommes intéressés à réaliser les mesures suivantes :

1. charge d'un processeur,
2. bande passante effective des liens de communication,
3. degré de obstruction du routage des messages sur la puissance réel de calcul des processeurs,
4. degré d'obstruction des calculs intensifs sur le temps de transfert des messages.

Ces mesures forment une base permettant de déduire d'autres mesures plus complexes, comme par exemple, le temps de transfert des messages par un protocole particulier sous diverses conditions de charge ausi bien de calcul que de communication en incluant plusieurs processeurs intermédiaires.

Mesure de la charge d'un processeur

La méthode consiste à mesurer le temps d'exécution d'un processus qui s'exécute tout seul sur un processeur. Ensuite, le même processus est exécuté en parallèle (en réalité en pseudo-parallélisme) avec d'autres processus, qui chargent le processeur, et on mesure à nouveau le temps d'exécution. La première mesure est divisée par la seconde pour obtenir la moyenne du temps d'unité centrale disponible sous la charge imposée.

Le processus qui réalise les mesures est toujours actif mais il peut être interrompu par un autre processus. Pour obtenir une meilleure précision, il faut exécuter ce processus en haute priorité : l'horloge haute priorité des transputers est 64 fois plus précis que celle de basse priorité.

Pour imposer une charge l durant une période T, il faut utiliser le processeur pendant q et se bloquer pendant w. Alors : $T = q + w$ et $l = \frac{q}{q+w}$.

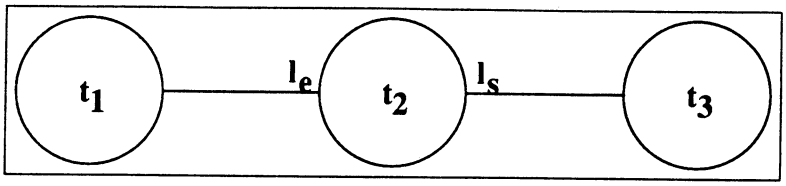


Figure 9.2: Disposition de processeurs pour mesures

Mesure de la bande passante effective des liens de communication

Cette mesure peut se réaliser en envoyant des données d'un processeur à son voisin. Le processeur récepteur est toujours prêt à recevoir des données qui arrivent par le lien utilisé. Le temps est mesuré dans le processeur émetteur, on prend la valeur de l'horloge juste avant d'envoyer la premier donnée et juste après l'envoi de la dernière.

Effet de la communication sur la puissance de calcul

Pour réaliser cette mesure il faut trois processeurs, connectés de la façon illustrée par la figure 9.2. Le processeur t_2 réalise un certain calcul, de façon répetitive, le temps de calcul est mesuré sans aucun autre processus actif dans t_2. Pour mesurer l'influence des activités de routage, le processus t_2 achemine des messages envoyés de t_1 à t_3 avec une fréquence variable.

Les processus associés au routage sont en haute priorité, ainsi, lorsqu'un message arrive à t_2 par le lien l_e, il réveille le processus *récepteur* bloqué en attente d'une communication par ce lien, le message est pris en charge jusqu'à être réémis sur le lien l_s. Nous sommes intéressés en mesurer comment l'activité de calcul est affectée par cette activité de routage.

Effet du calcul sur les communications

Pour cette mesure, il faut simplement inverser le rôle de la communication et du calcul, maintenant on fixe une certaine fréquence des messages entre t_1 et t_3 en mesurant le temps dans le cas où t_2 ne réalise aucun calcul, ensuite on réalise des activités de calcul dans t_2 et on mesure leur effet sur le temps de communication.

Deux problèmes se posent, le premier concerne la mesure du temps de communication, on ne peut réaliser la même mesure que dans le cas de la bande passante. On peut, par contre, mesurer seulement dans t_2 le temps entre le moment où le message arrive et quitte le processeur, qui sera affecté par les activités de calcul.

9.5 Conclusion

Ce chapitre a été consacré à l'examen de certains aspects de la mise en œuvre de la communication entre processus dans un réseau de transputers, ce que fait partie des travaux pratiques réalisés pour valider les aspects conceptuels concernant essentiellement la communication entre processus dans un ordinateur massivement parallèle.

Nous n'avons pas voulu décrire les détails de la programmation, notre démarche a été de souligner les aspects les plus remarquables de la mise en œuvre et de l'expérience obtenue.

En ce qui concerne les mesures de performance, nous avons préféré montrer ce qui constitue un ensemble de mesures de base qui peuvent être utilisées pour caractériser le temps de communication et surtout des mesures d'obstruction et d'interférences introduites entre le routage des messages et le calcul proprement dit. Les résultats et conclusions sur les mesures sont montrés dans l'annexe C.

Chapitre 10

Conclusions et perspectives

Bilan

L'idée centrale de cette thèse était l'étude et la réalisation d'un noyau de communication pour les ordinateurs multi-processeurs sans mémoire commune, avec application aux architectures reconfigurables Supernodes. Dans ce type d'architecture les performances sont très dépendantes de l'efficacité des communications, ce qui nous a conduit à privilégier l'étude des divers facteurs liés au contrôle de celle-ci.

Le contrôle correct et efficace de la communication n'est pas seulement une fonction de l'architecture cible, il requiert aussi une définition consistente des besoins de communication. Ces besoins sont définis, dans notre cas, par les mécanismes d'interaction entre processus du noyau de système d'exploitation parallèle Parx, qui est bâti par dessus le noyau de communication mise en œuvre dans cette thèse. Le modèle de processus de Parx comporte trois entités, bien adaptées aux paradigmes de programmation parallèle des architectures de machines visées, qui ont besoin de mécanismes de communication différents. Ces mécanismes, étudiés au chapitre 4, ont été pris en compte lors de la conception du noyau de communication, notamment en ce qui concerne les protocoles de communication développés au chapitre 8.

La conception et la mise en œuvre d'un noyau de communication pour des architectures multi-processeur sans mémoire commune fut l'objet des parties III et IV de cette thèse. Les problèmes liés à la communication furent clairement identifiés, et des solutions furent examinées, notamment en ce qui concerne le routage des messages sans interblocage. Nous avons proposé un algorithme original qui fut spécialement étudié pour être efficace et bien adapté aux besoins particuliers des architectures visées.

Au chapitre 8, nous avons proposé une structure générique pour l'interprétation des protocoles utilisés dans Parx : elle accepte simultanément différents protocoles dans la même architecture du noyau. Un ensemble de protocoles peut s'insérer modulairement en fonction des besoins ; les interfaces furent bien définies et les contraintes de correction établies. Nous avons aussi réalisé une étude approfondie d'un protocole de communication particulier, mais très important pour le modèle de système et les architectures qui nous intéressent : c'est le protocole synchrone, point à point, base du modèle d'interaction entre processus communicants. Nous avons examiné les aspects conceptuels et pratiques de la réalisation correcte de ce protocole pour un ensemble quelconque de processus communicants qui s'exécutent sur un réseau quelconque de processeurs.

Le noyau de communication fut mis en œuvre sur différents environnements à travers le temps. Une première version appelée **COMKER**, fut développée en langage occam sous l'environnement TDS [GLM90]. Cette version a été utilisée par les divers partenaires du projet **ESPRIT SUPERNODE**. Des versions plus récentes ont été écrites en langage C parallèle et le noyau complet a été intégré dans Parx, qui a été à son tour intégré dans PAROS, un système d'exploitation pour les machines parallèles sans mémoire commune développé dans le cadre du projet **ESPRIT SUPERNODE II**.

Des études visant les mesures de performances ont été entamés. Quelques mesures ont été présentées en annexe C mais il reste encore du travail à réaliser. D'une part il faut améliorer l'instrumentation. D'autre part, il faut encore réaliser des mesures, analyser leur validité et obtenir des conclusions par rapport au comportement du noyau.

Perspectives

Nous avons déjà remarqué que le développement du logiciel de base pour les architectures massivement parallèles est très en retard par rapport au matériel, par conséquent, la conception d'un tel logiciel doit inévitablement tenir compte de l'évolution du matériel. Nous sommes en train de présenter nos travaux au moment précis où des nouveaux processeurs sont annoncés, nous parlons précisément du transputer T9000 et du composant de communication C104 ; tous les deux annoncés par Inmos pour l'année 92. Ces processeurs incorporent des fonctionnalités de communication qui pourraient faire penser, de prime abord, que nos travaux seront rapidement dépassés. Or, ils viennent réaffirmer les idées exprimées tout au long de cette thèse, aussi bien concernant les communications que les modèles de processus pour la gestion adéquate du parallélisme massif.

Nous avons analysé les caractéristiques préliminaires de ces processeurs au cours du chapitre 6, et nous pouvons conclure que nos concepts, aussi bien en ce qui concerne les communications que par rapport au modèle d'exécution, ne sont pas mis en cause, l'apparition de processeurs tels que le T9000 ne fait que confirmer nos idées et ne peut que signifier l'amélioration des performances lors de sa mise en œuvre, à cause d'une participation plus importante du matériel dans les fonctions de routage. Tous nos concepts, et même nos développements logiciels, pourront être aisément repris avec le nouveau processeur.

La tendance actuelle de passer le maximun de fonctionnalités au niveau matériel nous fait penser que l'étape qui suit la maîtrise du routage de messages par le matériel consiste à intégrer les protocoles de communication. La structure générique proposée dans le chapitre 8 de cette thèse est bien adapté et constitu un point de départ des études visant une telle intégration.

La même optique s'applique au modèle d'exécution ; les concepts proposés forment une base générale pour la construction d'un noyau de système d'exploitation pour les architectures multi-processeurs sans mémoire commune, qui doivent se maintenir indépendamment du niveau auquel certaines opérations se réalisent. Nous pensons notamment à la gestion de la mémoire et à l'ordonnancement de processus qui sont actuellement intégrés par certains processeurs.

Nous espérons valider tous nos concepts par les étapes d'évaluation déjà entamées. D'autre part, les travaux théoriques menés actuellement au sein de l'équipe SYMPA, sur divers problèmes concernant les systèmes d'exploitation parallèles : gestion des objets, gestion de la mémoire, gestion des Entrées/Sorties, devront compléter Parx.

Annexe A

Files d'attente multiples

Nous présentons ici une étude sur le fonctionnement des files d'attente. Elle est inspiré des travaux réalisés au sein de l'équipe **SYMPA** par F. Grardel et C. Fabre en tant qu'étudiants de la troisième année **ENSIMAG 1989**, et qui ont contribué à la mise en œuvre du protocole occam réparti.

On fournit de primitives permettant de créer et de manipuler des files d'attente bloquantes et non bloquantes, avec plusieurs niveaux de priorités pouvant être modulées afin d'éviter l'introduction de famine. Une file bloquante est caractérisée par le fait que le processus qui insère un élément dans la file (le client) est bloqué jusqu'à ce que l'élément est servi. La file se charge de réveiller le processus client, pour cela, le pointeur sur l'espace de travail du client doit être maintenu dans la file.

En règle générale, le noyau de communication a besoin de files d'attente un peu partout. Le besoin de files multi-niveaux (multi-clients) avec priorité se fait sentir notamment dans les cas suivantes :

Dans le routage des messages :

Lorsqu'on veut donner la priorité au routage des messages par rapport aux messages locaux qui entrent au réseau dans un processeur donné, la **file** associé à chaque **émetteur** correspond à une file à deux niveaux de priorités.

Dans les protocoles :

Lorsqu'on souhaite réaliser un **émetteur** qui fournit un service prioritaire au **récepteur** par rapport au **serveur**, on a besoin d'une file à deux niveaux de priorités.

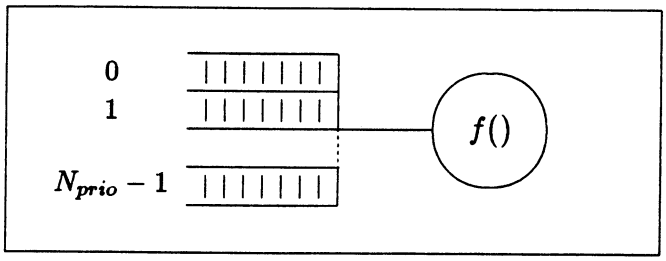


Figure A.1: Modèle de service à file d'attente multiple

La figure A.1 est une illustration d'une file à multi-niveaux de priorités, nous nous intéressons à la gestion des priorités qui permet de servir tout élément de l'ensemble de files au bout d'un certain temps. $f()$ représente le service appliqué à l'élément en tête de la file.

Les paramètres qui caractérisent la gestion de priorités sont :

P le nombre de priorités de la file, 0 étant la plus haute, $(P-1)$ la plus basse.

N permet de contrôler le comportement de la file afin d'assurer l'absence de famine entre les différents niveaux de priorité.

La gestion de la priorité

Si le paramètre N passé à la création de la file est égal à `FULL_PRIO`, la file fonctionne en mode prioritaire normal :

Règle 1 *Aucun élément de priorité p ne peut être servi tant qu'il reste des éléments de priorité $< p$ à servir.*

Il y a alors risque de famine. En effet, s'il arrive toujours dans la file des éléments de priorité $< p$, les éléments des priorités $\geq p$ ne seront jamais servis.

Pour assurer l'absence de famine, il faut un mécanisme permettant de servir aussi des éléments de priorité $\geq p$ même si il reste des éléments de priorités $< p$.

Si $N \neq$ `FULL_PRIO` alors $N > 0$ et on l'interpréte de la manière suivante :

Règle 2 *Au lieu de servir le $(N+1)^{ème}$ élément de priorité p, on essaye de servir (selon cette même règle), un élément de priorité $p+1$*

Soit α_n^p le $n^{ème}$ élément de la file de priorité p. Cherchons Θ_n^p, le nombre maximum de α_i^j qui seront servis avant que α_n^p le soit.

Plaçons nous pour cela dans le cas le plus défavorable : il y a toujours des élément à servir dans les files de priorités $< p$. Posons θ_n le nombre de α_i^j servis entre deux possibilités de service d'éléments de la file de priorité p. θ_p est égal au nombre d'éléments servis entre α_1^p (inclus) et α_2^p (exclus).

D'après la règle N° 2 du mode de gestion des priorités, nous avons :

$$\theta_0 = 1$$

Essayons de calculer θ_p par rapport à θ_{p-1}. Pour cela faisons un schéma des files de priorité p et $p-1$:

$\cdots$	α_{2N+1}^{p-1}	α_{2N}^{p-1}	$\cdots$	α_{N+2}^{p-1}	α_{N+1}^{p-1}	α_N^{p-1}	$\cdots$	α_2^{p-1}	α_1^{p-1}
$\cdots$	α_{2N+1}^{p}	α_{2N}^{p}	$\cdots$	α_{N+2}^{p}	α_{N+1}^{p}	α_N^{p}	$\cdots$	α_2^{p}	α_1^{p}

α_{N+1}^{p-1} viens de passer son tour pour que α_1^p puisse être servi, nous allons sommer le nombre de services nécessaires pour que α_2^p soit servi.

$$\theta_p = \underbrace{1}_{\alpha_1^p} + \underbrace{\theta_{p-1}}_{\alpha_{N+1}^{p-1}} + \cdots + \underbrace{\theta_{p-1}}_{\alpha_{2N}^{p-1}} + \underbrace{(\theta_{p-1} - 1)}_{\substack{\text{jusqu'à } \alpha_{2N+1}^{p-1} \\ \text{qui a passé son tour pour } \alpha_2^p}}$$

$$\theta_p = \underbrace{1}_{\alpha_1^p} + \underbrace{\theta_{p-1} + \cdots + \theta_{p-1}}_{N\text{fois}} + (\theta_{p-1} - 1)$$

$$\theta_p = (N+1)\theta_{p-1}$$

$$\text{soit :} \qquad \theta_p = (N+1)^p$$

Essayons maintenant d'exprimer Θ_n^p en fonction de θ_n.

- Il y a (n-1) éléments de priorité p à servir avant de servir α_n^p.
- La file de priorité p à laissé passé son tour $\lfloor \frac{n}{N} \rfloor$ fois.

- Entre le dernier moment où l'on a essayé de servir un élément de priorité p, et le moment où on viens pour servir α_n^p, $(\theta_{p-1} - 1)$ éléments ont été servis.

Ainsi, on obtient :

$$\boxed{\Theta_n^p = (n + \lfloor \tfrac{n}{N} \rfloor)(N+1)^p - 1}$$

Cette manière de gérer l'absence de famine est simple, les temps d'attente en fonction de la priorité ne pouvant être qu'en suite géométrique, mais elle convient parfaitement aux besoins du noyau, qui ne nécessite, en général, que deux niveaux de priorité.

choix du paramètre P

Dans les files utilisées pour les liens de sortie, le paramètre P indique l'importance que l'on accorde au routage sur les paquets locaux. On peut imaginer de réduire P pour les processeurs les plus sollicités par le routage, de façon à ne pas "trop" pénaliser les processus locaux.

Les files bloquantes

Ce type de file n'est utilisé que pour enfiler des entités actives, les processus[1]. Ce qui suit est une description succincte des opérations applicables aux files bloquantes.

Création d'une nouvelle file La création d'une file se fait par :

```
t_block_queue *new_block_queue(f,arg_static,P,N) ;
```

On reçoit en retour un pointeur vers un `t_block_queue`. Ce pointeur est l'identificateur de la file ainsi créée.

Introduction d'un élément dans la file cette opération se réalise par :

```
t_word enqueue_block(t_block_queue *q , t_objet *objet) ;
```

`objet` étant l'objet sur lequel on veut faire appliquer la fonction $f()$.

[1] On ne peut pas "bloquer" un objet inactif.

La fonction de service de la file La fonction $f()$ assurant le service est de la forme :

```
t_word f  (    const  void *   arg_static,
                      void *   objet,
               const  proc_id  client_Wdesc
          ) ;
```

Les éléments de ces files étant des processus, ils sont chaînés par les champs **nextp** de leurs *workspace*. Les processus sont réveillés après que leur **objet** ait été traité, et le résultat renvoyé par $f()$ est retourné par **enqueue_block**.

Les files non bloquantes

Ce type de file n'est utilisé que pour enfiler des objets passifs, tels que les paquets ; elles sont typiquement utilisées par les processus associés au routage, la fonction $f()$ n'existe pas. Les opérations de base sont les suivantes :

Création d'une nouvelle file La création d'une file se fait par :

```
t_noblo_queue *new_noblo_queue(owner,arg_static,P,N) ;
```

On reçoit en retour un pointeur vers un **t_noblo_queue**. Ce pointeur est l'identificateur de file.

Introduction d'un nouvel élément dans la file Ceci est réalisé par :

```
t_word enqueue_noblo(t_noblo_queue *q , t_objet *objet) ;
```

Cette opération doit inclure le réveil du processus **owner** dans le cas où la file était vide. Dans le cas des listes circulaires, le champ **liste_last** du **owner** doit être mis à jour.

retrait d'un élément de la file Seulement **owner** peut réaliser cette opération :

```
t_objet dequeue_noblo(t_noblo_queue *q) ;
```

Retourne l'adresse de l'objet se trouvant en tête de la file, dans le cas des listes circulaires il doit mettre à jour le champ **liste_first** dans la structure associée à **owner**.

Annexe B

L'alternative Occam sur transputer

Nous ne reprenons ici que les aspects du transputer qui nous intéressent et qui concernent directement notre problème, le lecteur davantage intéressé sur le transputer peut lire par exemple [Inmos86].

Le transputer a été conçu pour exécuter des programmes occam. Il dispose donc de primitives assembleur qui réalise, sur le silicium, des algorithmes propres à ce type de communication et à la gestion des processus qui découle.

Dans toute la suite, les valeurs entre `MIN_INT` et `MIN_INT+3` représentent des adresses invalides sur le transputer (de `0x80000000` à `0x80000003`).

B.1 Gestion des processus sur le transputer

Les processus transputers sont connus par un pointeur sur leur espace de travail : *workspace*. Ce *workspace* est un pointeur vers un zone mémoire de 6 mots[1] de long, décrivant le processus.

Parmi ceux-ci, deux champs nous intéressent :

`nextp` est utilisé par le micro-séquenceur pour chaîner les processus entre eux, ce champ est libre lorsque le processus n'est ni actif (il n'utilise pas l'unité centrale), ni activable (il attend pour l'unité centrale), ni en attente sur des événements du temps.

[1] Le mot est de la taille d'une adresse : 4 octets sur des T414 et T800, 2 octets sur un T212.

`status` contient l'information relative à l'état du processus par rapport aux *rendez-vous* :

`MIN_INT+1` *début d'alternative*, le processus entre dans une alternative, il est en train de chercher, parmi tous les gardes ayant leur `condition` à vrai, tous les canaux qui sont prêts pour une communication. Si le canal est prêt pour communiquer alors il contient une valeur > `MIN_INT` + 3.

`MIN_INT+2` *attente en alternative*, le processus est prêt pour une alternative, mais aucun des canaux n'est présent au *rendez-vous*. Il est en attente.

`MIN_INT+3` *fin d'alternative*, le processus est actif ou activable, et au moins un *rendez-vous* est possible dans l'alternative.

toute autre valeur indique que le processus est présent au *rendez-vous*, cette valeur étant considérée comme l'adresse du message à transférer.

B.2 Le *rendez-vous* inconditionnel

Le transputer possède deux instructions assembleur implémentant l'*entrée* : (?), et la *sortie* : (!) ; elles sont appelées `In` et `Out` respectivement.

Soit deux processus communiquant de manière inconditionnelle par un canal de nom `canal`[2] :

`ResetChan(canal)` Cette instruction renvoie la valeur contenue dans le canal, et met `MIN_INT` dans le canal.

`In(canal,adresse,longeur)` Cette instruction réalise effectivement l'*entrée*, à savoir :

Si la valeur du canal est `MIN_INT`, le processus est arrivé le premier au *rendez-vous*, il place donc son *workspace* dans le canal et sauvegarde l'adresse où il veut recevoir le message dans le champs `status` de son *workspace*, et se bloque.

Si la valeur du canal est différente de `MIN_INT`, le processus prend cette valeur comme étant le *workspace* de son correspondant, effectue le transfert du message, réveille son correspondant, et continue son exécution.

`Out(canal,adresse,longueur)` Cette instruction met en œuvre la *sortie*. Pour le *rendez-vous* inconditionnel, elle fonctionne exactement comme le `In`. Nous examinerons plus tard son comportement dans le cas d'une alternative.

[2]Le canal est un mot mémoire de la taille d'une adresse.

Pour un *rendez-vous* inconditionnel, les instruction `In` et `Out` sont symétriques. On remarque que dans le cas où le `In` et le `Out` n'auraient pas défini la même longueur, on ne peut savoir quelle sera la longueur effective du message échangé : c'est le processus qui arrive en deuxième au *rendez-vous* qui fixe la longueur.

B.3 L'alternative

Aux trois états des processus transputers, présentés dans la section B.1, correspondent des instructions de machine spécifiques permettant un codage efficace des alternatives, que nous décrivons succinctement dans les paragraphes suivants. Une description complète des instructions machine du transputers se trouve dans [Inmos87]

`AltBegin` Positionne le champ *workspace*.`status` du processus courant à *début d'alternative.*

`EnableChan(canal,cond)` Si `cond` est vrai, examine l'état du `canal` :

Si la valeur du contenu du `canal` est `MIN_INT`, le correspondant n'est pas encore au *rendez-vous*, la valeur du *workspace* courant est mise dans le canal.

Sinon le correspondant est déjà au *rendez-vous* (`canal` contient le *workspace* du correspondant) le champ *workspace*.`status` du processus courant est positionné à *fin d'alternative.*

Si `cond` est false, on ne fait rien.

`AltWait` Si le champ *workspace*.`status` est égal à *fin d'alternative*, le processus courant continue en séquence.

Sinon il s'endort et sera réveillé par l'un des processus émetteurs impliqués dans l'alternative.

`DisableChan(cond,canal,code)` Si `cond` est vrai, examine l'état du `canal` :

Si le canal contient son propre *workspace*, le correspondant de ce canal n'est toujours pas arrivé : l'état du canal est mis à `MIN_INT` .

Si le canal contient autre chose, c'est le *workspace* du correspondant de ce canal qui signale ainsi sa présence au *rendez-vous.* Une adresse indiquant où se trouve le code de cette branche de l'alternative est sauvegardée.

Si `cond` est false, on ne fait rien.

`AltEnd` Le programme fait un saut vers la dernière branche de code qui a été sauvegardée par un `DisableChan` ; chacune de ces branches débutant par le code effectuant le `In` sur son canal.

`Out(canal,adresse,longueur)` N'est donc pas aussi simple que nous l'avons indiqué pour le *rendez-vous* inconditionnel ; il faut maintenant vérifier le champ **status** du processus se trouvant dans **canal** et agir en conséquence. Pour exprimer les actions, mettons nous à la place du processus qui exécute l'intruction `Out`.

Si **status** contient *début d'alternative*, on le passe à *fin d'alternative* pour signaler au récepteur qu'au moins un des correspondants est au *rendez-vous*, puis on s'endort.

Si **status** contient *attente en alternative*, on se met dans le canal et on réveille le processus bloqué sur l'alternative qui va pouvoir prendre sa decision et faire un `In`, puis on s'en dort.

Si l'état indique *fin d'alternative*, on se met juste dans le canal et on se bloque : le processus récepteur est en train de résoudre son alternative.

Si ce n'est aucune de ces valeurs, alors le correspondant nous attend pour un *rendez-vous* inconditionnel, on effectue le transfert de données, on le réveille et on continue notre exécution.

Il faut juste faire remarquer que dans la cas du *rendez-vous* inconditionnel c'est le dernier processus qui prend la décision de communiquer indépendamment de s'il réalise un `In` ou un `Out`. Dans le cas de l'alternative par contre, c'est toujours le processus récepteur faisant le `In` qui prend la décision de communiquer, un processus émetteur ne peut que faire noter sa présence sur le canal.

La figure B.1 en est une illustration du code généré par l'alternative.

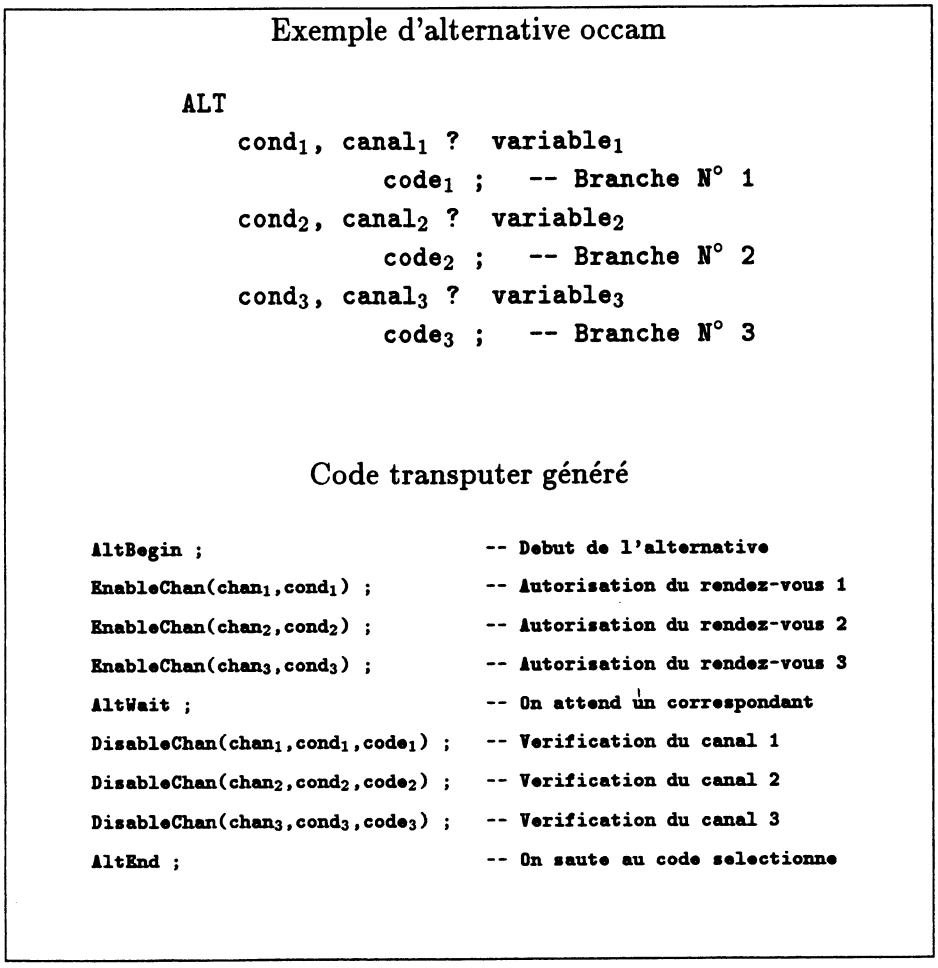


Figure B.1: Mise en œuvre de l'alternative en occam et code transputer généré.

Annexe C

Mesures des performances du noyau de communication

L'étude de performances dans les architectures parallèles est un sujet encore peu exploré et les paramètres à mesurer ne sont pas encore clairement établis. On se contente généralement avec les mesures de bande passante sur les liens de communication et des délais de transfert des messages. Pour cette raison nous allons nous intéresser aussi à la méthodologie des mesures de performance. Notre but est donc d'obtenir un ensemble d'outils de mesure, validés convenablement, qui nous permettront d'évaluer l'impact de différents paramètres du noyau de communication. Nous mettons donc en avant la méthode employée et la validation des instruments de mesure[1].

Les résultats présentés ici ont pour seul objectif de mieux comprendre le fonctionnement réel du noyau. Le prototype actuel du noyau n'a pas été développé dans le but de produire un code optimum, mais plutôt de permettre un large éventail d'expérimentations. Les mesures de performance doivent permettre l'amélioration de ce prototype.

Les mesures ont été effectuées sur des Supernodes comportant seize transputers de travail T800 et un contrôleur T414 fonctionnant à 20 Mhz. La vitesse des liens de communication était de 10 Mbits/s.

Nous examinons par la suite la mise au point de l'outil de mesure de la charge d'un processeur, puis de l'outil permettant d'imposer une charge à un processeur.

[1] Cette étude fut préparée en étroite collaboration avec Y. Langué, une présentation similaire de l'étude se trouve donc dans [Langue91]

C.1 Mesure de la charge d'un processeur

La mesure de la charge d'un processeur est assez immédiate. Nous avons choisi de mesurer d'abord le temps d'exécution d'un processus isolé, puis d'effectuer la même mesure avec d'autres processus s'exécutant sur le même processeur. Le pourcentage de temps processeur disponible sous la charge des processus ajoutés s'obtient par simple division des deux valeurs.

Le processus de mesure effectue une simple boucle de façon à être constamment à la fois activable et interruptible. Nous utilisons largement les caractéristiques de l'ordonnancement des processus sur le transputer, qui limite les points de préemption pour expiration de tranche de temps à quelques instructions bien définies. Pour une mesure de charge précise, ce processus doit être exécuté en haute priorité.

C.2 Imposition d'une charge au processeur

La solution adoptée pour imposer une charge l sur une période de temps T, consiste à effectuer une attente active de q unités de temps, puis attendre w unités de temps. Nous avons les relations suivantes :

$$\left.\begin{array}{l} T = q + w \\ l = \frac{q}{q+w} \end{array}\right\} \qquad \text{(C.1)}$$

C.2.1 Choix des paramètres

La période T:

Elle doit être suffisamment courte pour permettre d'influer sur des processus s'exécutant pendant des durées très brèves avant de se bloquer. Prenons l'exemple d'un programme qui réalise une boucle comportant un traitement très bref (quelques microsecondes), suivi d'une communication. Si le temps de traitement est suffisament faible par rapport à T, le programme peut exécuter plusieurs boucles sans être affecté par le processus de charge. Le quantum pour l'ordonnancement des processus transputers varie entre 1024 et 2048 μsecondes, ce qui donne une borne supérieure à T.

Le temps d'exécution q:

Nous parlerons aussi du "quantum" d'exécution par analogie avec le quantum d'exécution imposé par un ordonnanceur à un processus. Notre objectif est

d'obtenir une charge correspondant à l'effet d'un processus application. Le quantum devrait donc correspondre à celui observable pour les applications parallèles. Cette valeur est très variable en fonction des applications. De plus, les processus de haute priorité perturbent la formule C.1 car, lors de leur exécution, ils peuvent allonger indéfiniment la valeur de w. Cela conduit à une imprécision de la valeur de charge imposée. Une solution serait d'allonger la période T, mais les processus à temps d'exécution très petit ne subiraient alors pas de charge.

Les résultats de mesures sont donnés à ± 1 tick d'horloge, ce qui signifie une incertitude absolue de 1 μseconde en haute priorité, et de 64 μsecondes en basse priorité.

Dans une première expérience, nous faisons exécuter le processus de charge en même temps qu'un processus dont nous voulons observer le comportement, et nous mesurons ses temps d'attente w. Les deux processus s'exécutent en haute priorité. La figure C.1 illustre les variations de w en fonction du temps pour $q = 1024$ μsecondes et $l = 50$ %. Nous constatons un pic pour w avec une périodicité de $7 \times T$. Ceci indique que le processus observé n'est pas affecté par la charge artificielle que nous imposons. L'explication est qu'il s'exécute avec une période supérieure à T (environ $7 \times T$), et chaque fois pour un temps bien supérieur au quantum de charge q, soit entre 10000 et 30000 μsecondes.

La figure C.2 montre l'effet de la charge imposé sur les processeurs de basse priorité. On constate que les processus en basse priorité sont mesurés avec précision par notre dispositif, sur une plage de charges imposées importante. Lors de cette mesure, la charge s'est stabilisée après deux périodes du processus de charge, la valeur de T étant choisie à 20480 μsecondes, soit un temps de stabilisation d'approximativement 41 millisecondes. Une stabilisation encore plus rapidet a été observés pour des valeurs de q plus faibles ($q = 512$, $q = 256$).

La même expérience a été effectuée en faisant varier le quantum d'exécution du processus observé, afin d'établir la plage de validité de notre observation. Les figures C.3 et C.4 représentent leurs résultats. La charge observée est exprimée en fonction du quantum d'exécution du processus observé, pour une charge imposée de 75%. Les résultats mesurés pour de faibles valeurs de quanta du processus observé présentent des variations importantes par rapport à la charge souhaitée. Ceci s'explique par le fait que les valeurs mesurées dans ces cas là sont faibles par rapport à la précision de mesure dont nous disposons. Par exemple, une mesure de 7 périodes d'horloge avec une incertitude absolue d'une période aboutit à une incertitude relative de plus de 14%, ce qui rend la mesure peu significative. Les mesures suivantes, effectuées avec des valeurs de q plus faibles ($q = 512$, 256 et

128 μsecondes) montrent que la précision de la charge imposée se dégrade lorsque le quantum du processus de charge diminue. L'explication est que le temps de commutation de contexte commence à devenir significatif lorsque les quantités mesurées sont faibles. Cette remarque contredit les observations effectuées pour des processus de haute priorité, pour lesquels de faibles valeurs de q accéléraient la convergence de la valeur de la charge imposée. Nous constatons donc un comportement différent entre les processus selon qu'ils sont observés lors de leur exécution en haute ou en basse priorité.

Ces mesures nous permettent de conclure qu'une correction est nécessaire lorsque les valeurs de q sont faibles. Elle doit prendre en compte le temps de commutation de contexte et le temps écoulé pendant les portions d'exécution du code du processus de charge non prises en compte dans le délai d'attente active. La correction doit aussi prendre en compte les variations que les processus de haute priorité peuvent introduire lors de l'estimation de w, ce qui a été illustré lors de la première expérience (figure C.1). Le noyau s'exécutant en haute priorité, nous sommes principalement intéressés par la mesure de processus de haute priorité.

Le tracé de la courbe des valeurs mesurées de w en fonction de différentes valeurs de q montre une variation quadratique. Une approximation de la courbe nous a permis d'effectuer une correction sur la valeur de w. Les résultats alors obtenus pour de faibles valeurs de q sont bien meilleurs. Pour de très faibles valeurs de q, les mesures sont mauvaises parce que le temps écoulé pendant le calcul de w et la correction appliquée devient non négligeable devant les valeurs de q. La valeur $q = 16$ μsecondes donne les meilleurs résultats.

Les figures C.5, C.6 et C.7 représentent la charge mesurée pour différentes valeurs de q pour des valeurs de consigne entre 5 et 95% de charge. La précision de notre mesure est très bonne dans une plage de durées d'exécution commençant à 403 μsecondes. Nous éviterons les mesures d'une durée inférieure à cette borne.

Nous avons ainsi obtenu un outil de mesure des processus de haute priorité, mais qui malheureusement n'est pas applicable aux processus de basse priorité, comme le montre la figure C.8. Remarquons sur cette figure que le pourcentage de temps CPU disponible est constant et en dessous de la valeur de consigne. Ceci s'explique parce que le calcul de w n'est plus perturbé par l'exécution d'autres processus, et les conditions qui nous amenés à la formule de charge ne sont plus vérifiées. Nous utiliserons donc le processus de charge que nous avons présenté au début de cette section pour les processus de basse priorité. Notons aussi, comme il est illustré par la figure C.9, que les variations du quantum du processus chargeur, appliqué aux processus de basse priorité, n'influent pas sur la valeur

de la charge imposée. Nous ne pouvons espérer utiliser le même outil avec un quantum plus élevé.

Nous disposons maintenant d'un outil de charge d'un processeur, capable d'imposer une charge de consigne avec une incertitude relative inférieure à 5%.

C.3 Influence du nombre de processeurs intermédiaires

La figure C.10 montre le débit des messages, en fonction du nombre de processeurs intermédiaires, pour différentes tailles de messages. Pour éviter le problème de la synchronisation des horloges, les messages sont retournés par le destinataire à l'émetteur. Ceci permet de lire la même horloge. Il suffit ensuite de diviser le délai écoulé par deux. Des processeurs voisins ont un nombre nul d'intermédiaires. Aucun des processeurs n'exécutait de programme application lors de notre expérience. Nous constatons que le débit est une fonction croissante de la taille des messages échangés.

C.3.1 Influence de la charge du processeur sur la communication

La figure C.11 montre le débit en fonction du nombre de processeurs intermédiaires alors que chaque processus subit une charge de 75%. Nous pouvons constater que le débit diminue d'approximativement 12%. La figure C.12 montre la même mesure sous une charge de 25%. Le débit ne diminue que de 5.7%. Le débit n'est donc pas une fonction linéaire de la charge des processeurs. Les liens fonctionnant en parallèle avec le processeur de traitements, un accroissemet de charge influe uniquement sur les traitements effectués par le noyau. Remarquons aussi que la variation est plus sensible pour les messages de taille importante.

La figure C.13 montre l'influence de la charge du processeur sur le routage. L'expérience menée comporte trois processeurs connectés en pipeline. L'un des processeurs extrêmes envoie des messages aussi vite qu'il le peut à l'autre extrêmité. Le processeur intermédiaire se contente d'acheminer les messages vers leur destination. Il supporte aussi un processus de charge, les autres n'en supportent pas.

Nous pouvons voir que les résultats sont meilleurs que lors de l'expérience avec tous les processeurs subissant une charge. De 70% à 95%, soit une augmentation de 25% de charge, la variation de débit correspondante est seulement de 6.2%. A nouveau, le débit n'est pas une fonction linéaire de la charge. De plus, re-

marquons que le débit de bout en bout n'est pas notablement affecté lorsque la charge des processeurs augmente.

C.3.2 Influence du routage sur les programmes utilisateurs

L'expérience visait à mesurer le pourcentage de temps processeur utilisé par le noyau, et donc au détriment des processus application, pour une activité de routage maximale. Cette activité est obtenue en utilisant au maximum la bande passante des liens. Pour des processus application s'exécutant en haute priorité, nous avons constaté une baisse de 29% du temps processeur qui leur serait accordé sans le noyau. un processus de basse priorité subirait sans doute une baisse plus importante.

C.4 Conclusion

L'objectif de l'étude était d'une part de mettre en place des outils et une méthode de mesures. d'autre part de réaliser quelques mesures préliminaires permettant de clarifier les paramètres importants du noyau de communication. Nous avons pu constater que la mise au point des outils de mesure est assez difficile, et que leur mauvaise qualité peut avoir des répercussions très importantes sur les résultats des mesures. Le champs de mesures de performance du noyau Parx reste largement ouvert et l'expérimentation doit être poursuivie.

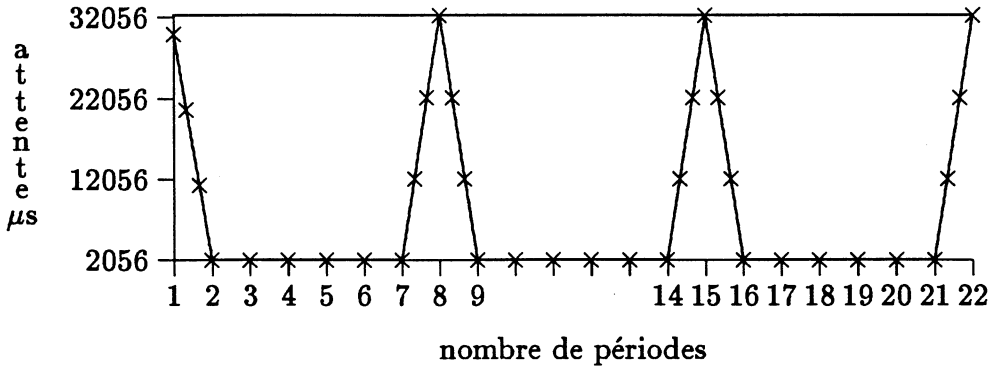


Figure C.1: variations du temps d'attente w, pour $q = 1024\mu s$, $l = 50\%$

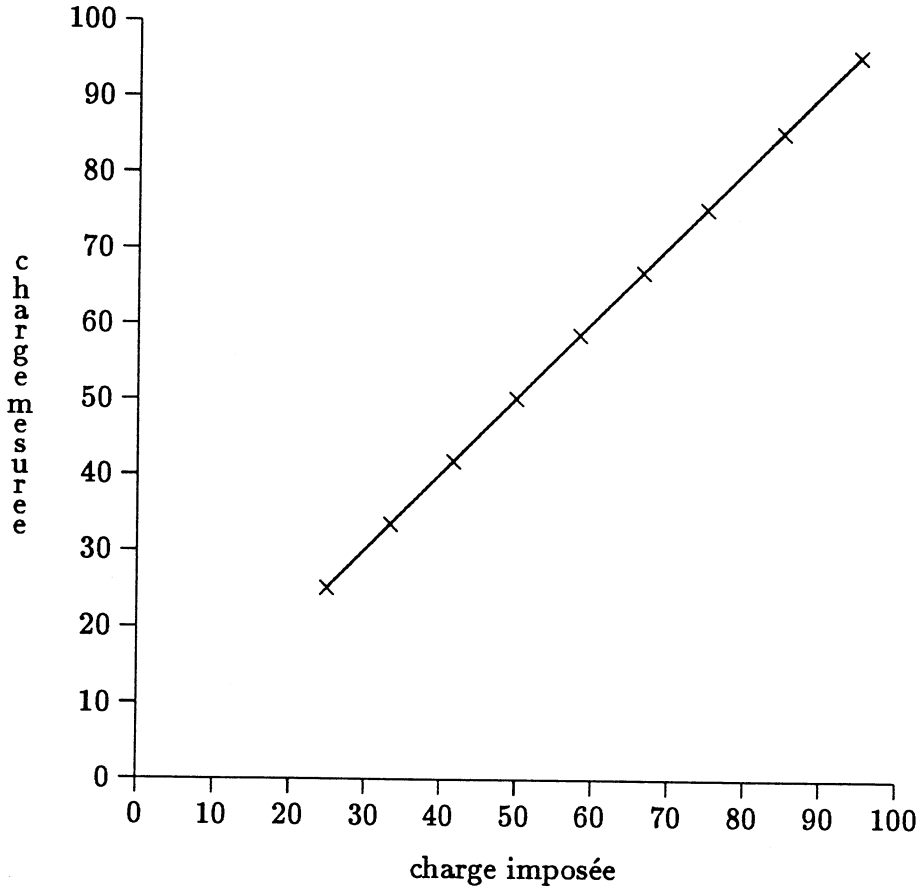


Figure C.2: mesure de l'effet de la charge imposée sur les processus de basse priorité, $q = 1024\mu$s, $l = 25, 50, 75\%$

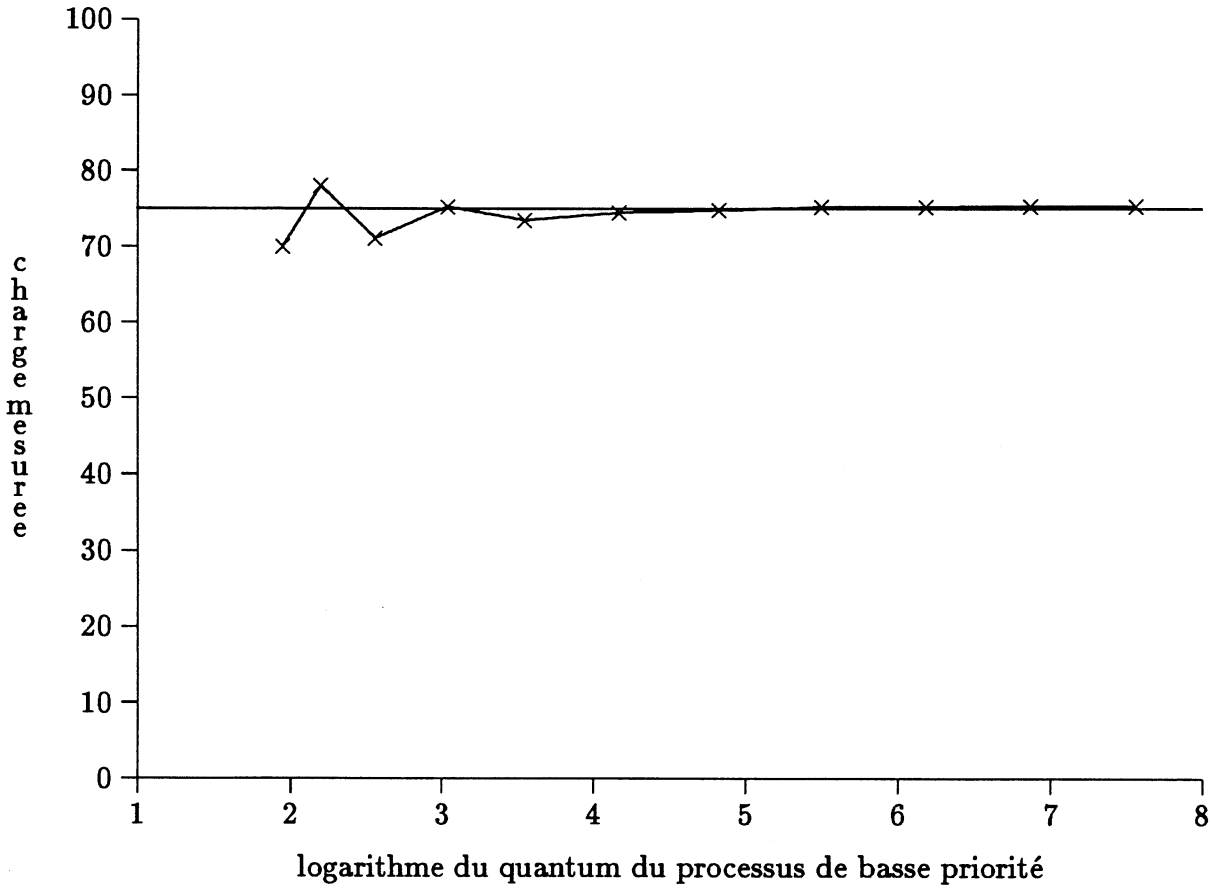


Figure C.3: effet de la charge imposée pour différentes valeurs du quantum du processus de basse priorité mesuré, $q = 1024\mu s$, $l = 75\%$

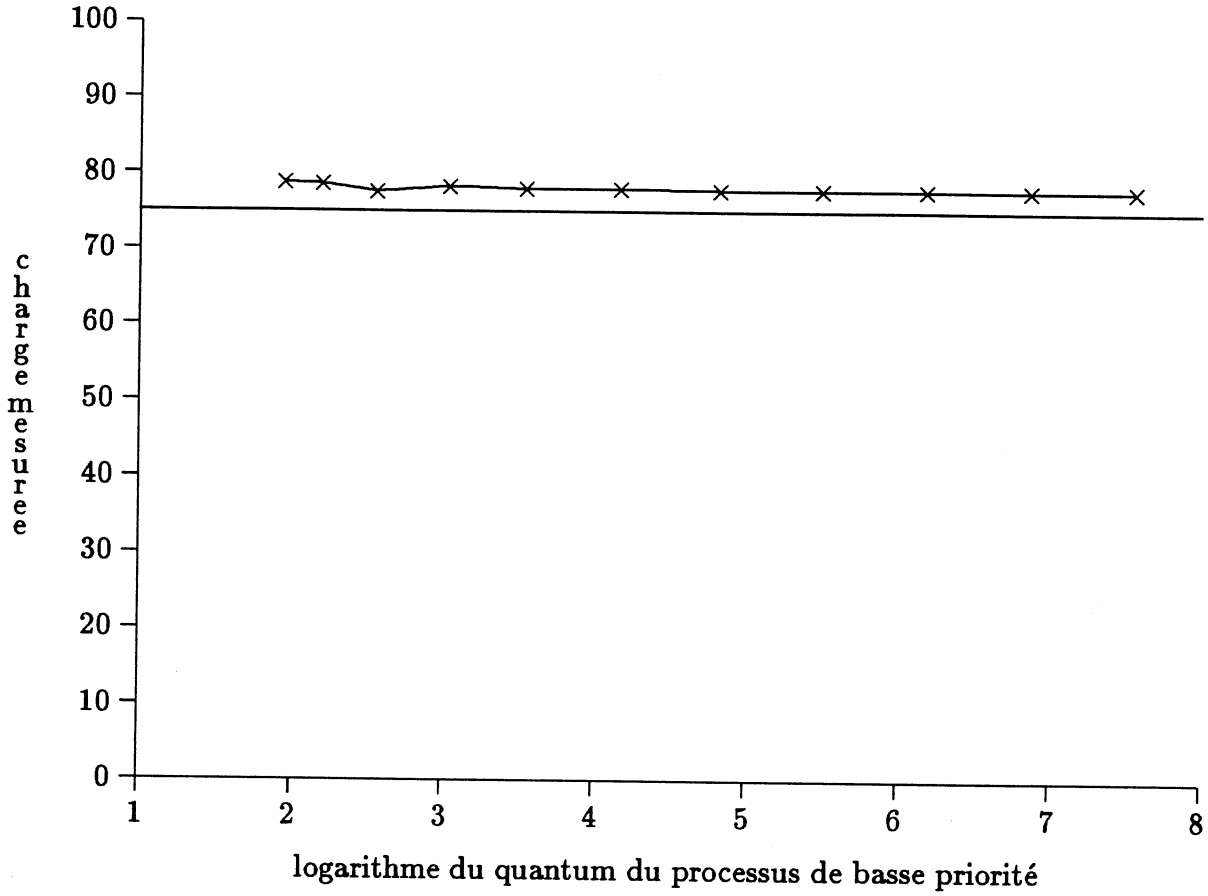


Figure C.4: effet de la charge imposée pour différentes valeurs du quantum du processus de basse priorité mesuré, $q = 128\mu s$, $l = 75\%$

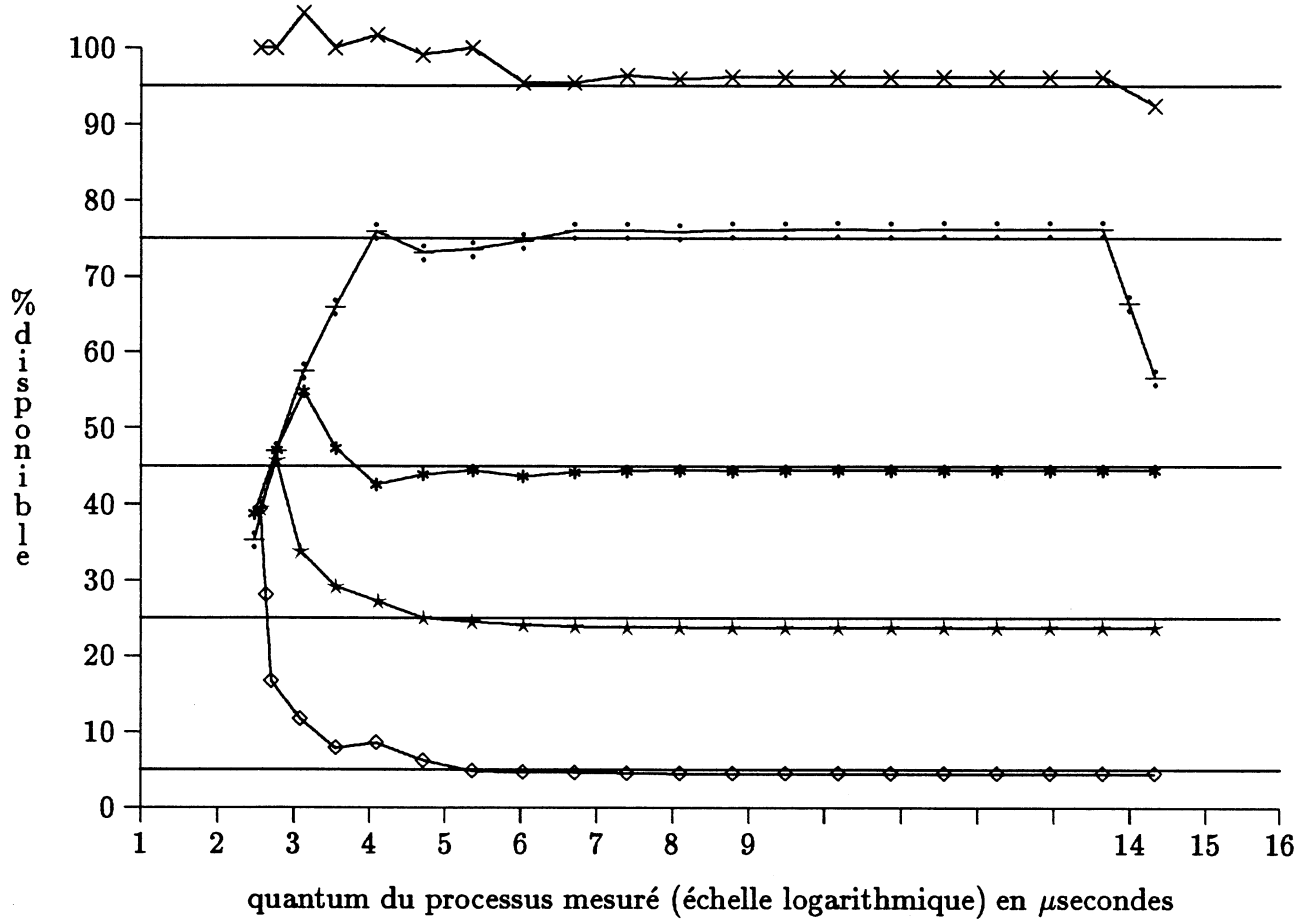


Figure C.5: pourcentage de processeur disponible aux processus de haute priorité en fonction du quantum du processus observé, 5, 25, 45, 75 et 95%

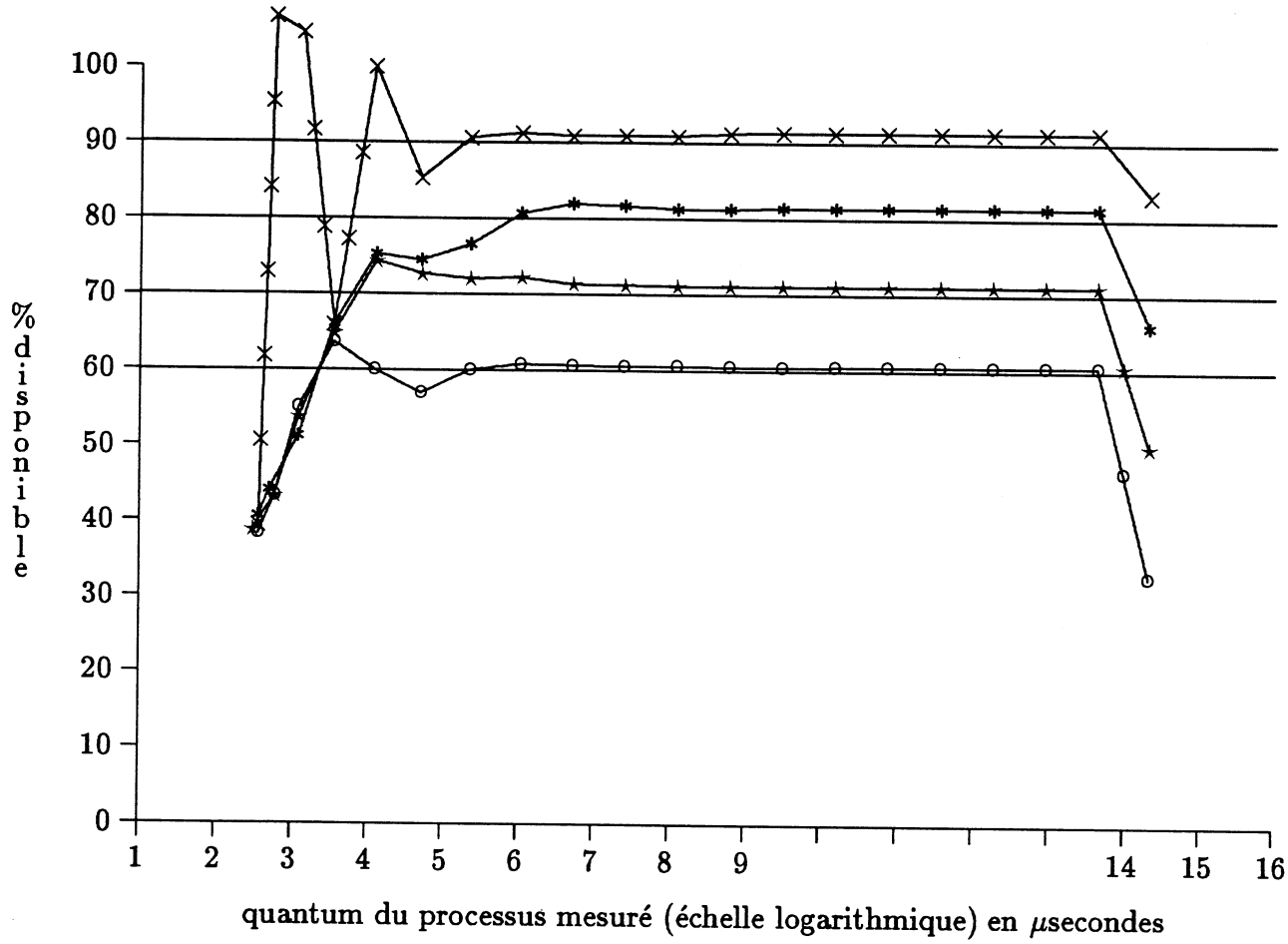


Figure C.6: pourcentage de processeur disponible aux processus de haute priorité en fonction du quantum du processus observé, 60, 70, 80 et 90%

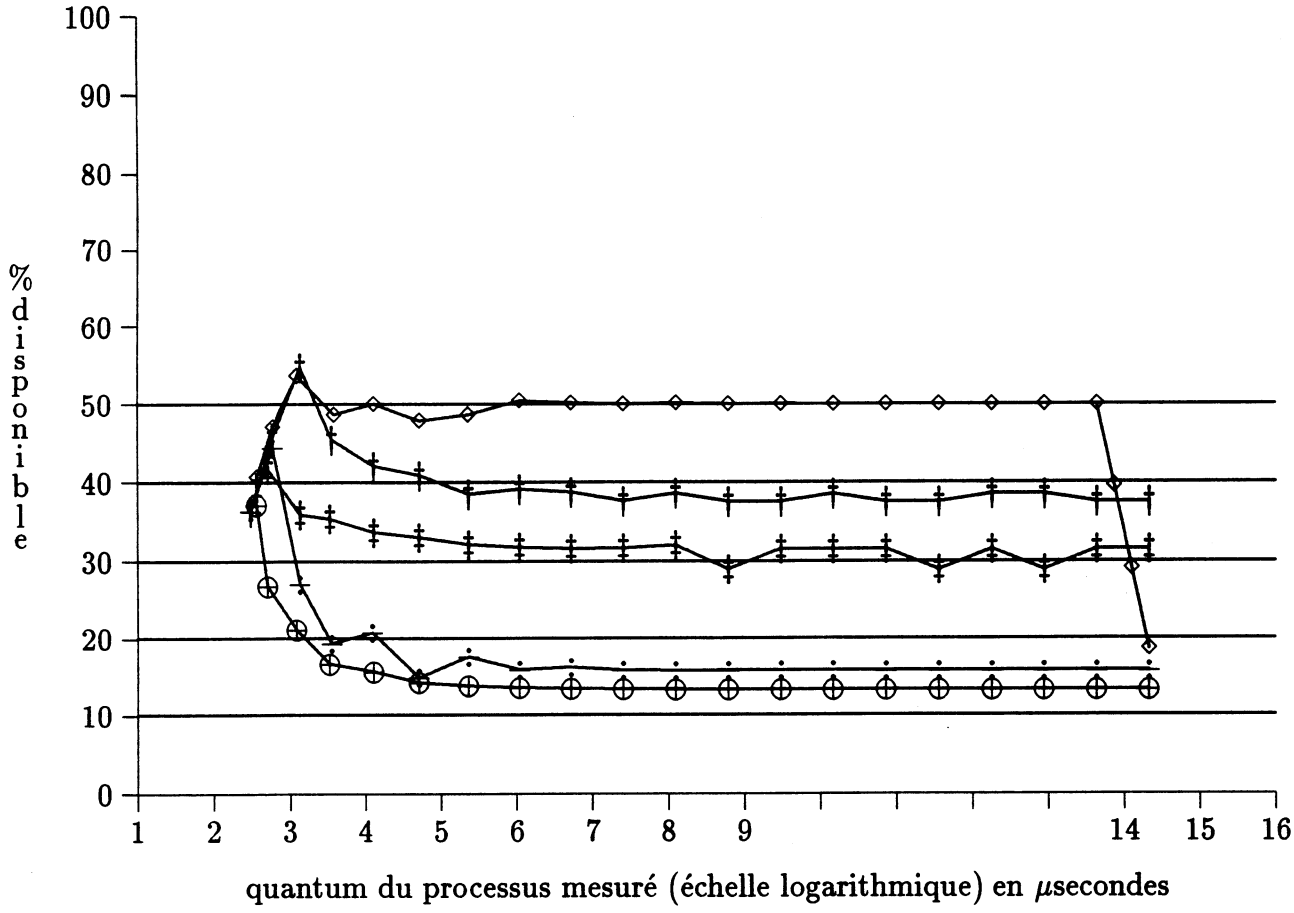


Figure C.7: pourcentage de processeur disponible aux processus de haute priorité en fonction du quantum du processus observé, 10, 20, 30, 40 et 50%

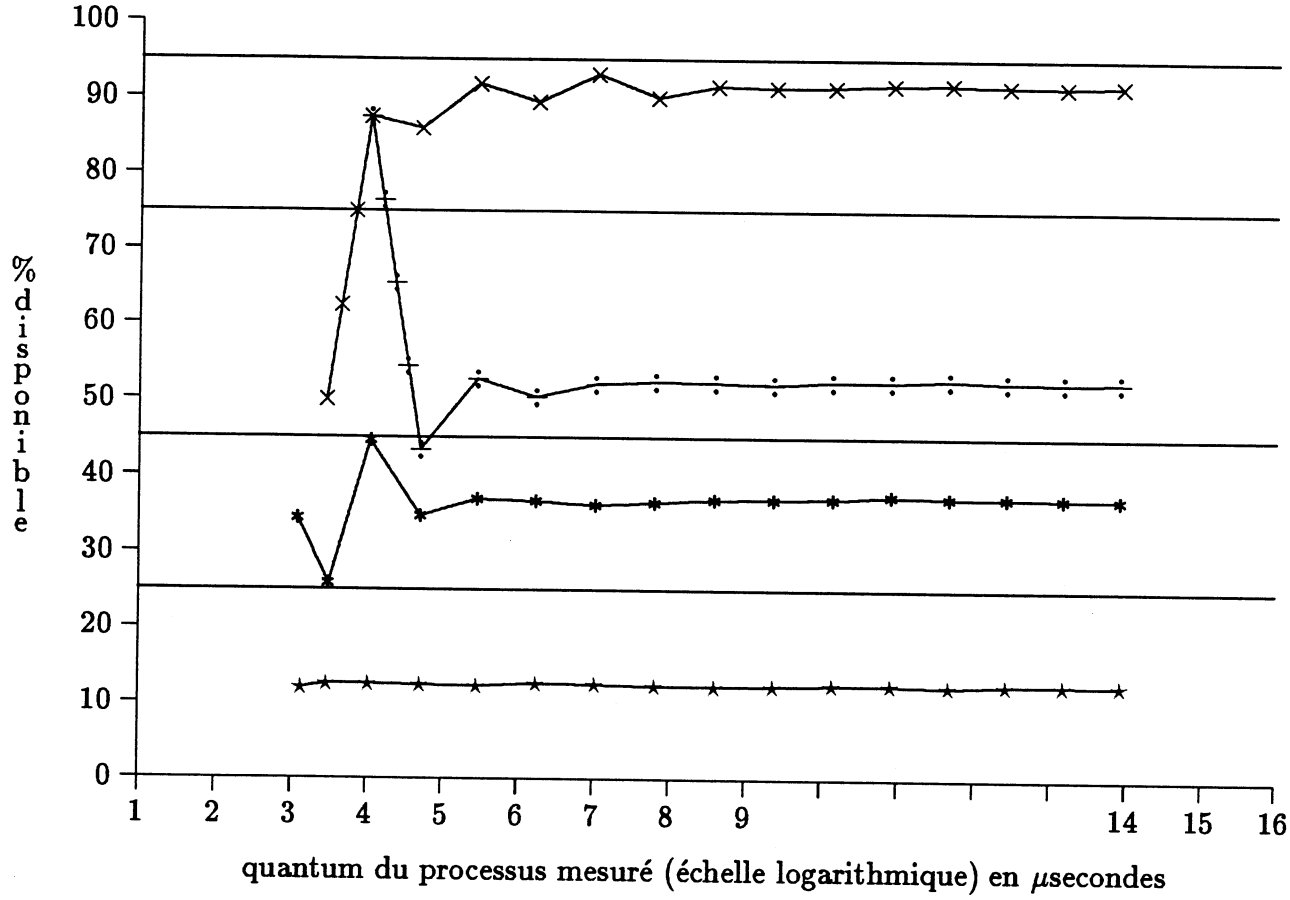


Figure C.8: pourcentage de processeur disponible aux processus de basse priorité en fonction du quantum du processus observé, 25, 45, 75 et 95%

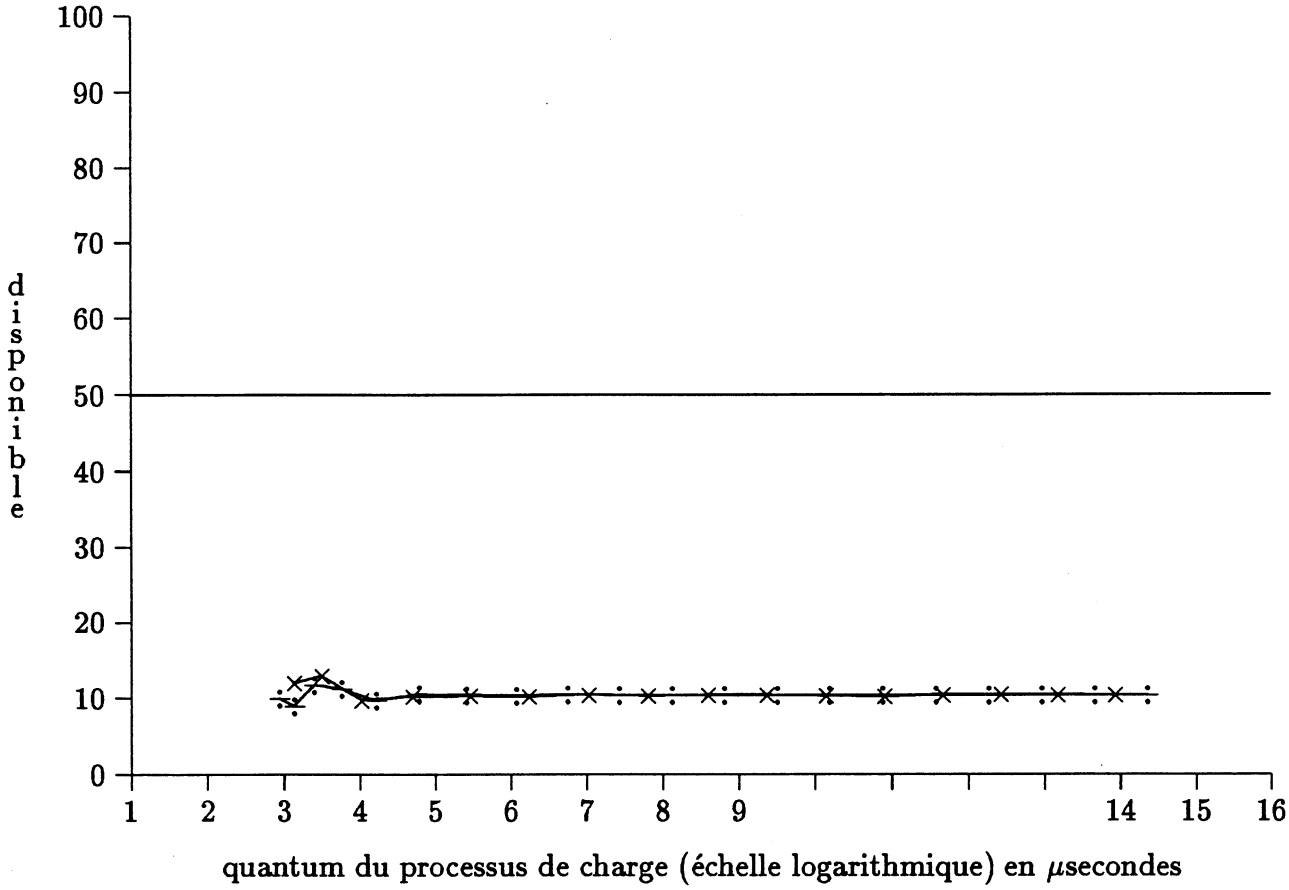


Figure C.9: charge mesurée pour un processus de basse priorité, en faisant varier le quantum du processus de charge, $l = 50\%$

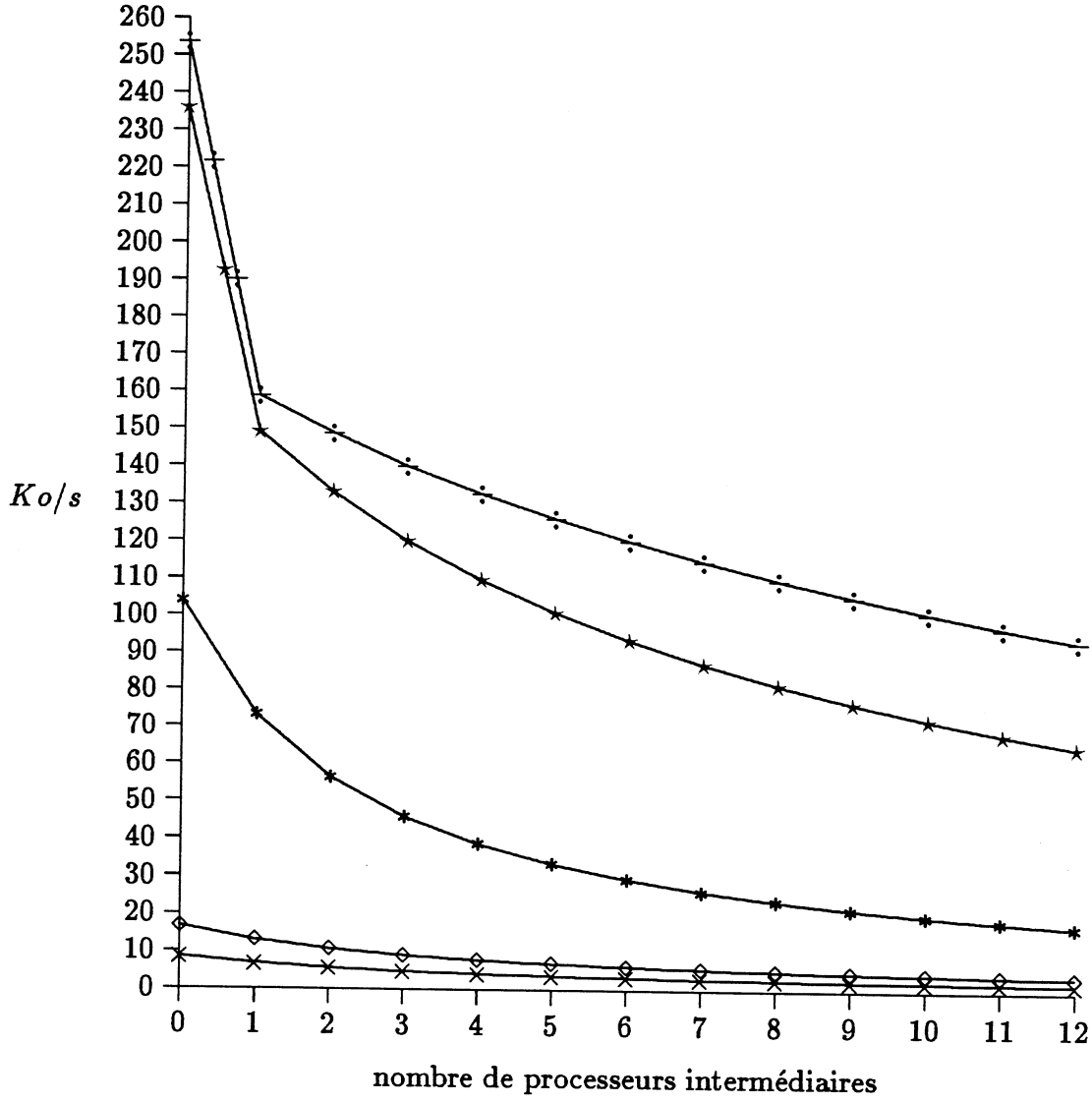


Figure C.10: Débit sous une charge uniforme de 0%

× pour les messages de 16 octets

◇ pour les messages de 32 octets

∗ pour les messages de 256 octets

⋆ pour les messages de 2048 octets

÷ pour les messages de 4096 octets

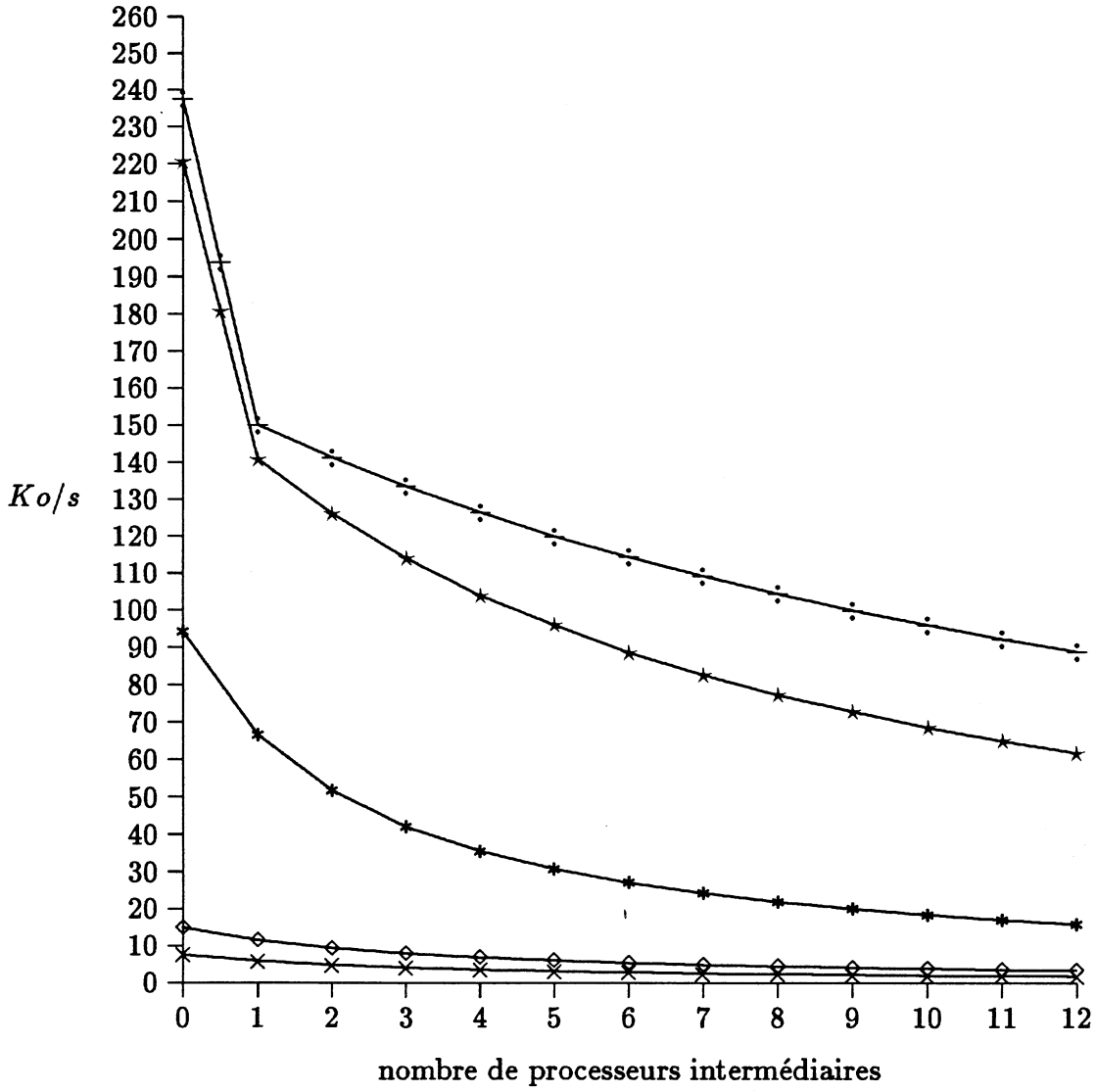


Figure C.11: Débit sous une charge uniforme de 75%

× pour les messages de 16 octets

◇ pour les messages de 32 octets

∗ pour les messages de 256 octets

⋆ pour les messages de 2048 octets

÷ pour les messages de 4096 octets

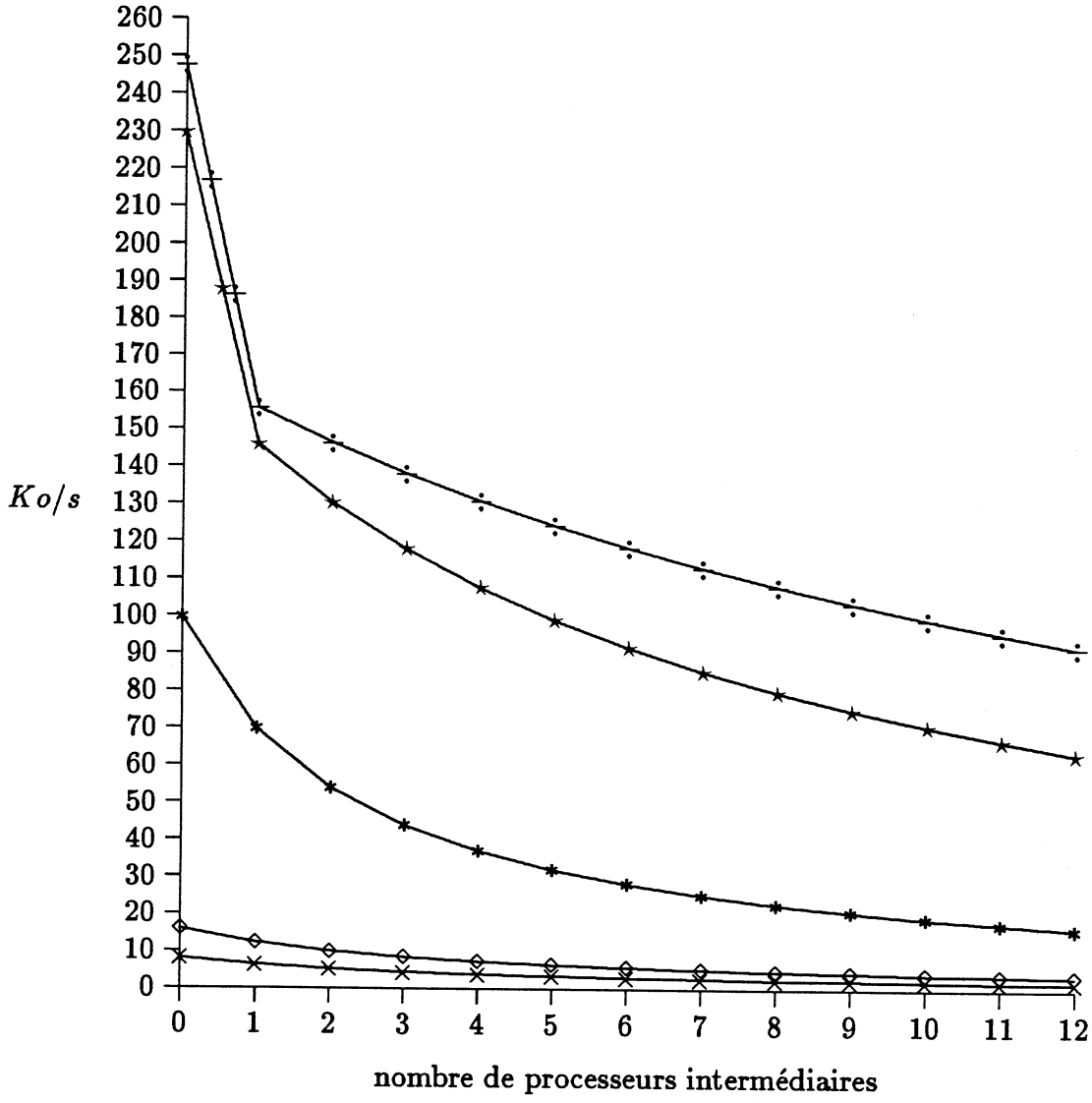


Figure C.12: Débit sous une charge uniforme de 25%

× pour les messages de 16 octets

◇ pour les messages de 32 octets

∗ pour les messages de 256 octets

⋆ pour les messages de 2048 octets

÷ pour les messages de 4096 octets

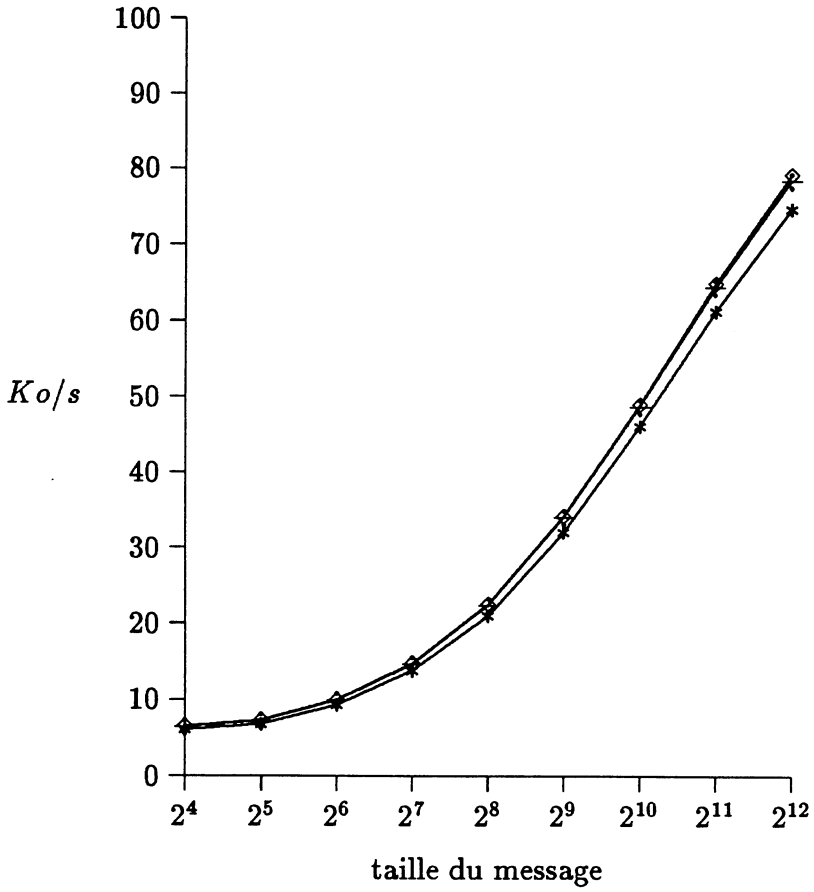


Figure C.13 : ralentissement de la communication par un processus de charge, 75, 85 et 95% de charge

÷ pour une charge de 75%

◇ pour une charge de 85%

∗ pour une charge de 95%

Bibliographie

[Accet86] Mike Accetta et al. *MACH: A New Kernel Foundation for UNIX Development.* Computer Science Department, Carnegie Mellon University. (cité dans la section 5.4)

[AGHR89] François Armand, Michel Gien, Frédéric Herrmann and Marc Rozier. *Revolution 89 or "Distributing UNIX Brings it Back to its Original Virtues".* Chorus Systèmes, 1989. (cité dans les sections 3.4.2 et 5.3)

[AgJan86] Dharma P. Agrawal and Virendra K. Janakiram. *Evaluating the Performance of Multicomputer Configurations.* IEEE COMPUTER, May 1986. (cité dans la section 1.3.1)

[AHU74] Alfred v. Aho, John E. Hopcroft and Jeffrey D. Ullman. *The Design and Analysis of Computer Algorithms.* Addison-Wesley, 1974. (cité dans la section 7.7)

[Ahu79a] V. Ahuja. *Routing and flow control in Systems Network Architecture.* IBM System Journal, Vol. 18, $N°$ 2, 1979. (cité dans les sections 6.2 et 6.6)

[Ahu79b] V. Ahuja. *Algorithm to Check Network States for deadlocks.* IBM Journal res. develop. Vol. 23, $N°$ 1, January 1979. (cité dans la section 6.2)

[AlGot89] G. S. Almasi and A. Gottlieb. *Highly Parallel Computing.* The Benjamin/Cummings Publishing Company, Inc. 1989. (cité dans la figure 1.1 à la page 32, et dans les sections 2.2 et 2.3)

[AnPa88] F. André et J-L. Pazat. *Le placement de tâches sur des architectures parallèles.* Technique et Science Informatiques, Vol. 7, $N°$ 4, 1988. (cité dans la section 3.4)

[AnTwi87] J. K. Annot and R. A. H. van Twist. *A novel Deadlock Free and Starvation Free Packet Switching Communication Processor.* LNCS $N°$ 258, June 1987, 68-85. (cité dans la section 7.3.2)

[ARSha89] Vadim Abrossimov, Marc Rozier and Marc Shapiro. *Generic Virtual Memory Management for Operating System Kernels.* Chorus Système, September 1989. (cité dans les sections 5.3 et 5.4)

[AtSe88] William C. Athas and Charles L. Seitz. *Multicomputers: Message-passing Concurrent Computers.* IEEE COMPUTER, August 1988. (cité dans la section 1.3.1)

[Badal86] D. Z. Badal. *The Distributed Deadlock Detection Algorithm.* ACM Transactions on Computer Systems, Vol. 4, $N°$ 4, November 1986. (cité dans la section 7.1)

[Barnes68] Barnes et al. *The ILLIAC IV Computer.* IEEE Transsaction on Computers 17, $N°$ 8, August 1968. (cité dans la section 1.3.1)

[Batch80] K. Batcher. *Design of a Massively Parallel Processor.* IEEE Transactions on Computers 29, $N°$ 9, September 1980. (cité dans la section 1.2.1)

[BBG87a] Jacek Blażewics, Jerzy Brzeziński and Giorgio Gambosi *Time-Stamp Approach to Store-and-forward Deadlock Prevention.* IEEE Transactions on Communications, Vol. COM-35, $N°$ 5, April 1987. (cité dans la section 7.3.2)

[BCS89] A. Billionnet, M-C. Costa et A. Sutter. *Les problèmes de placement dans les systèmes distribués.* Technique et Science Informatiques, Vol. 8, $N°$ 4, 1989. (cité dans la section 3.4)

[Benes] Voir [WuFeng84] pp. 117-125. (cité dans la section 1.2.1)

[BGM86] P.M. Behr, W.K. Giloi and H. Mühlenbein. *SUPRENUM: The German Supercomputer. Projet-Rationale and concepts.* IEEE Int. Conf. on Parallel Processing, 1986. (cité dans la section 5.5)

[BiNe84] A. D. Birell and B. J. Nelson. *Implementing Remote Procedure Calls.* ACM Transaction on Comp. Syst. February 1985, 39-85. (cité dans la section 8.3)

[Black90] David L. Black. *Scheduling Support for Concurrency and Parallelism in the Mach Operating System.* IEEE COMPUTER, May 1990. (cité dans les sections 3.3, 3.4.1, 3.6 et 5.4)

[Bork89] Shekhar Borkar et al. *iWarp: An Integrated Solution to High-Speed Parallel Computing.* Rapport CMU-CS-89-104, CMU, 11 Janv. 1989. (cité dans la section 2.4)

[Briat89] J. Briat et al. *PARX: a parallel operating system for transputer-based machines.* Applying transputers dased parallel machines, OUG-10, Ed. by André Bakkes, April 1989, Enschede, Netherland. (cité dans la présentation de la thèse)

[BroRo84] S.D. Brookes and A.W. Roscoe. *Deadlock Analysis in Network of Communicating Processes.* Technical Report CMU-CS-84, Carnegie Mellon University, 1984. (cité dans la section 7.1)

[BRT87] H.E. Bal, R. van Renesse ans A.S. Tanenbaum. *Implementing Distributed Algorithms Using Remote Procedure Calls.* Proc. National Computer Conference, AFIPS, 1987. (cité dans la section 8.3)

[Bryan88] Oliver A. McBryan. *New architectures: Performance highlights and new algorithms.* Parallel Computing 7 (1988) 477-499. (cité dans la section 1.4)

[Bryan89] Oliver A. McBryan. *Current Developments in Parallel Computation.* Technical Report CU-CS-433-89, University of Colorado, April 1989. (cité dans les sections 1.1 et 2.7)

[BSTan89] H. E. Bal, J. G. Steiner and A. S. Tanenbaum. *Programming Languages for Distributed Computing System.* ACM Computing Surveys, Vol. 21, $N°$ 3, September 1989. (cité dans la section 3.3)

[CaGel89l] N. Carriero and D. Gelernter. *Linda in Contex.* Communications of the ACM, Vol. 32, $N°$ 4, April 1989. (cité dans la section 3.5.1)

[CaGel89p] N. Carriero and D. Gelernter. *How to Write Parallel Programs : A Guide to the Perplexed.* ACM Computing Surveys, Vol. 21, $N°$ 3, September 1989. (cité dans la section 3.2)

[Castro91] Harold Castro. *Gestion de ressources virtuelles partagées dans les machines parallèles.* Rapport DEA, LGI-IMAG, Grenoble, September 1991. (cité dans la section 8.9)

[CaVi88] Ugo De Carlini and Umberto Villano. *A simple Algorithm for Clock Synchronization in Transputer Networks.* Software-Practice and Experience, Vol. 18(4), April 1988. (cité dans la section 6.7)

[CDB79] W. Chou, J.D. Powell and A.W.Bragg, Jr.. *Comparative Evaluation of Deterministic and Adaptive Routing.* Flow Control in Computer Networks, IFIP, North-Holland Publishing Company (1979). (cité dans la section 6.4)

[ChaMa84] Jo-Mei Chang and N.F. Maxemchuk. *Reliable Broadcast Protocols.* ACM Transactions on Computer Systems, Vol. 2, $N°$ 3, August 1984. (cité dans la section 8.3)

[ChaMi79] K.M. Chandy and J. Misra. *Deadlock Absence Proofs for Networks of Communicating Processes.* Information Processing Letters, Vol. 9, $N°$ 4, November 1979. (cité dans la section 7.1)

[ChaYu87] Cheung-Wing Chan and Tak-Shing P. Yum. *An Algorithm for Detecting and Resolving Store-and-Forward Deadlocks in Packet Switched Networks.* IEEE Transactions on Communications, Vol. COM-35, $N°$ 8, August 1987. (cité dans la section 7.3.1)

[Chor89] CHORUS Kernel V3.1: Specification and Interface. Chorus Système, April 1989. (cité dans la section 5.3)

[CJS87] Israel Cidon, Jeffrey M. Jaffe and Moshe Sidi. *Distributed Store-and-Forward Deadlock Detection and Resolution Algorithms.* IEEE Transactions on Communications, Vol. COM-35, $N°$ 11, November 1987. (cité dans la section 7.3.1)

[Cmmm] voir [AlGot89] pp. 420.

[CORNA81] Systèmes informatiques répartis. GRUPE CORNAFION. Dunod Informatique, 1981. (cité dans les sections 7.3.2 et 8.5)

[Crow85] W. Crowther et al. *The Butterfly (TM) Parallel Processor.* IEEE Computer Architecture Technical Committee Newsletter, September-December 1985. (cité dans la section 1.3.2)

[DaSe86] William J. Dally and Charles L. Seitz. *The Torus Routing Chip.* Distributed Computing (1986) 1:187-196. (cité dans les sections 1.3.1 et 6.2.2)

[DaSe87] William J. Dally and Charles L. Seitz. *Deadlock-Free Message Routing in Multiprocessor Interconnection Networks.* IEEE Transaction on Computers, Vol. C-36, $N°$ 5, May 1987. (cité dans les sections 7.3.2 et 7.4.1)

[DHN90] Mark Debbage, Mark Hill and Denis Nicole. *Virtuel Channel Router Deliverable.*
WP3, ESPRIT P2701, U. Of Southampton, October 1990. (cité dans la section 8.4)

[EMM89] J. Eudes, L. Mugwaneza et T. Muntean. *PDS: Advanced program development system for transputer based machines.* Proc. of the Occam User Group, IOS Amsterdam, April 1989. (cité dans la section 9.1)

[Eudes90] J. Eudes. *PDS : Un générateur des systèmes de développement pour machines parallèles.* Thèse Docteur INPG, LGI-IMAG, Grenoble, Novembre 1990. (cité dans la section 9.1)

[Fly72] Michel J. Flynn. *Some Computer Organizations and their Effectiveness.* IEEE Transactions on Computers, Vol. C-21, 1972. (cité dans la section 1.2)

[Garnett] N.H. Garnett. *HELIOS : An Operating System for the Transputer.* Perihelion Software Ltd. (cité dans la section 5.7)

[GCST90] E. Gallizzi, M Cannataro, G. Spezzano and D. Talia. *A Deadlock-Free Communication System for a Transputer Network.* OUG-12, Tools and techniques for transputer applications. Edited by Stephen J. Turner. IOS Press netherlands 1990. (cité dans la section 7.3.2)

[GeKle80] Mario Gerla and Leonard Kleinrock. *Flow Control: A Comparative Survey.* IEEE Transactions on Communications, Vol. COM-28, $N°$ 4, April 1980. (cité dans les sections 6.3, 6.6 et 7.3.2)

[Geler85] David Gelernter. *Generative Communication in Linda.* ACM Transactions on Programming Languages and Systems, Vol. 7, $N°$ 1, January 1985. (cité dans la section 3.5.1)

[Geler81] David Gelernter. *A DAG-Based Algorithm for Prevention of Store-and-Forward Deadlock in Packet Networks.* IEEE Transaction on Computers, Vol. C-30, $N°$ 10, October 1981. (Aussi dans : Performance of data communication systems and their applications, G. Pujolle (ed.), North Holland Publishing Company, 1981.) (cité dans la section 7.3.2)

[GF11] voir [AlGot89] pp. 325-331. (cité dans la section 1.2.1)

[GHKP78] A. Giessler, J.Hänle, A. Köning and E. Pade. *Free Buffer Allocation-An Investigation By Simulation.* Computer Networks 2 (1978) 191-208. (cité dans la section 7.3.2)

[GliSh80] Virgil D. GliGor and Susan H. Shattuck. *On Deadlock Detection in Distributed Systems.* IEEE Transactions on Software Engineering, Vol. SE-6, $N°$ 5, september 1980. (cité dans la section 7.1)

[GLM90] N. González, Y. Langué and T. Muntean. *A Communication Kernel for Netwok of Transputers: Virtualisation of distributed synchronous message passing.* Working Paper, ESPRIT PROJET 2528-SUPERNODE II, February 1990. (cité dans la section 9.2)

[Gonz88] Néstor González. *Noyau de communication pour le Supernode.* Rapport DEA, LGI-IMAG, Grenoble, September 1988. (cité dans la section 8.4)

[Gopal85] Inder S. Gopal. *Prevention of Store-and-Forward Deadlock in Computer Networks.* IEEE Transactions on Communications, Vol. COM-33, $N°$ 12, December 1985. (cité dans la section 7.3.2)

[Gott83] A. Gottlieb et al. *The NYU Ultracomputer-Designing an MIMD Shared Memory Parallel Computer.* IEEE Transactions on Computers, February 1983, 175-189. (cité dans la section 1.3.2)

[GraFa90] F. Grardel et C. Fabre. *Contrôle de la communication pour machine massivement parallèle.* Rapport 3-éme Année ENSIMAG, Grenoble, 27 Juin 1990. (cité dans les sections 8.4 et 9.3)

[Guill89] Marc Guillemont. *CHORUS: A Support for Distributed and Reconfigurable ADA Software.* Chorus Systèmes, October 1989. (cité dans la section 5.3)

[Günt81] Klaus D. Günther. *Prevention of Deadlocks in Packet-Switched Data Transport Systems.* IEEE Transactions on Communications, Vol. COM-29, $N°$ 4, April 1981. (cité dans les sections 6.3, 6.6 et 7.1)

[Haber69] A.N. Habermann. *Prevention of System Deadlocks.* Communications of the ACM, Vol. 12, $N°$ 7, July 1969. (cité dans la section 7.1)

[Hayes86] John P. Hayes et al. *A Microprocessor-based Hypercube Supercomputer.* IEEE MICRO October 1986. (cité dans la section 1.3.1)

[HiLu87] Peter A. J. Hilbers and Johan J. Lukkien. *Deadlock-Free Message Routing in Processor Networks.* Department of mathematics and computing science, University of Groningen, Netherlands, July 1987. (cité dans la section 7.3.2)

[Hoare78] C.A.R. Hoare. *Communicating Sequential Processes.* Communications of the ACM, Vol. 21, $N°$ 8, August 1978. (cité dans les sections 3.3 et 8.4)

[Hoare85] C.A.R. Hoare. *Communicating Sequential Processes.* Prentice-Hall, Englewoods Cliffs, 1985. (cité dans la section 3.3)

[Holt72] Richard C. Holt. *Some Deadlock Propierties of Computer Systems.* Computing Survays, Vol. 4, $N°$ 3, September 1972. (cité dans la section 7.1)

[Hwang87] Kai Hwang. *Advanced Parallel Processing with Supercomputers Architectures.* Proceedings of the IEEE, Vol. 75, $N°$ 10, October 1987. (cité dans la section 1.1)

[ICOT88] International Conference on Fifth Generation Computer Systems. FGCS'88, December 1988. (cité dans la section 2.7)

[Inmos86] IMS T800 Architecture. Technical note 6, INMOS Bristol, 1986. (cité dans la section B)

[Inmos87] The transputer instruction set – a compiler writers' guide. INMOS Corporation, February 1987. (cité dans la section B.3)

[Inmos88] Inmos Ltd. *Transputer Development System.* Prentice Hall, 1988. (cité dans la section 8.4)

[Irland78] Marek I. Irland. *Buffer Management in a Packet Switch.* IEEE Transactions on Communications, Vol. COM-26, $N°$ 3, March 1978. (cité dans la section 6.3)

[IsBor90] Petro Istavrinos and Lothar Borrman. *A process and memory model for a parallel distributed-memory machine.* Lecture Notes in Computer Science $N°$ 457. CONPAR90-VAPP IV, edited by H. Burkha. Zurich, September 1990. (cité dans les sections 3.4.2 et 5.6)

[JMY89] Jessope CR, Miller PR, Yantchev JT. *High Performance Communications in Processor Networks.* ACM Computer Architectures News, Vol. 17, $N°$ 7, June 1989.(cité dans la section 6.8)

[JoBi88] Thomas A. Joseph and Kenneth P. Birman. *Reliable Broadcast Protocols.* Department of Computer Science, Cornell University, June 1988. (cité dans la section 8.3)

[KeKle79] Paviz Kermani and Leonard Kleinrock. *Virtual Cut-Through: A New Computer Communication Swithching Technique.* Computer Networks 3 (1979) 267-286. (cité dans les sections 6.2.1 et 6.7)

[KeKle80] Paviz Kermani and Leonard Kleinrock. *A Tradeoff Study of Switching System in Computer Communications Networks.* IEEE Transactions on Computers, Vol. C-29, $N°$ 12, December 1980. (cité dans les sections 6.2.1 et 6.7)

[Klein76] Leonard Kleinrock. *Queueing Systems, Vol II: Computer Applications.* New York: Wiley Interscience 1976. (cité dans la section 6.2.1)

[Krako87a] Sacha Krakowiak. *Les systèmes d'exploitation répartis : évolution récente et tendances de la recherche.* Technique et Science Informatiques, Vol. 6, $N°$ 2, 1987. (cité dans les sections 3.3, 5.1 et 7.3.2)

[KTFB89] M.F. Kaashoek, A.S. Tanenbaum, S. Flynn Hummel and H.E. Bal. *An Efficient Reliable Broadcast Protocol.* Operating Systems Review, Vol. 23, October 1989. (cité dans la section 8.3)

[Lamp78] L. Lamport. *Time, clocks, and the ordering of events in a distributed system.* Communications of the ACM, 21, 558-565, 1978. (cité dans la section 6.7)

[Langue87] Yves Langué Tsobgny. *Un modèle pour la diffusion basé sur les ensembles de refus.* Rapport DEA, LGI-IMAG, Grenoble, September 1987. (cité dans la section 8.3)

[Langue91] Yves Langué Tsobgny. *Architectures de Systèmes d'Exploitation Parallèles : PARX.* Thèse Docteur INPG, LGI-IMAG, Grenoble, à paraître (1991). (cité dans les sections 4.1, 6.2.2, 8.1, 8.4)

[LeBe88] J. Legatheaux Martins et Y. Berbers. *La désignation dans les systèmes d'exploitation répartis.* Technique et Science Informatiques, Vol. 7, $N°$ 4, 1988. (cité dans la section 8.5)

[LeTa87] J. van Leeuwen and R.B. Tan. *Interval Routing.* The Computer Journal, Vol. 30, $N°$ 4, 1987. (cité dans les sections 6.4, 7.3.2 et 7.5)

[LM88] Yves Langué et Traian Muntean. *PARX: A Unix-like Operating System for Transputer-based Parallel Supercomputers.* Proc. Workshop on Unix and Supercomputers, Usenix, Pittsburg, PA, September 1988. (cité dans la présentation de la thèse)

[MEMT88] T. Muntean, J. Eudes, L. Mugwaneza and C. Tricot. *Operating (reconfigurable) Networks of Transputers.* OUG-7th, Parallel Programming of Transputer Based Machines, Traian Muntean (ed.), IOS Amsterdam Springfield, VA 1988. (cité dans la section 9.1)

[MMS90] L. Mugwaneza, T. Muntean and I. Sakho. *A deadlock-free routing algorithm with network size independent buffering space.* CONPAR90-VAPPIV, September 1990, Zurich. (cité dans la section 7.3.2)

[MS80a] Philip M. Merlin and Paul J. Schweitzer. *Deadlock Avoidance in Store-and-Forward Networks-I: Store-and-Forward Deadlock.* IEEE Transactions on Communications, Vol. COM-28, $N°$ 3, March 1980. (cité dans les sections 7.3.2 et 7.5)

[MuSa91] L. Mugwaneza T. Muntean et I. Sakho. *Algorithms de configuration des machines hiérarchiques Supernodes.* Rapport de Recherche IMAG, Mai 1991. (cité dans les sections 2.6 et 9.1)

[Mulder88] J.C. Mulder. *On the Amoeba Protocol.* Report CS-R8827 , July 1988 Computer Sc./Department of Software Technology, Centrum voor Wiskunde en Informatica (CWI), Netherlands. (cité dans la section 5.2)

[Mullen89] S.J. Mullender. *Amoeba: High performance distributed computing.* Report CS-R8937 , October 1989. Computer Sc./Department of Software Technology, Centrum voor Wiskunde en Informatica (CWI), Netherlands. (cité dans la section 5.2)

[Mullen90] S.J. Mullender et al. *Amoeba: A distributed Operating System for the 1990s.* IEEE COMPUTER, May 1990. (cité dans la section 5.2)

[Muntxx] Traian Muntean. *Supernode : Architecture Parallèle & Dynamiquement Reconfigurable de Transputers.* Onzièmes Journées francophones sur l'informatique. (cité dans la section 1.1)

[Munt88] Traian Muntean. *Les Supercalculateurs à Transputers.* La Recherche $N°$ 204, Vol. 19, Novembre 1988. (cité dans la section 3.2)

[Occam2] INMOS Occam 2 Reference Manual. Prentice-Hall, Englewoods Cliffs 1988. (cité dans les sections 3.3 et 8.4)

[PeUp89] David Peleg and Eli Upfal. *A Trade-Off between Space and Efficiency for Routing Tables.* Journal of the Association for Computing Machinery, Vol. 36, $N°$ 3, July 1989. (cité dans la section 7.8)

[PeSch75] Michael C. Pennotti and Mischa Schwartz. *Congestion Control in Store and forward Tandem links.* IEEE Transactions on Communications. Vol. COM-23, $N°$ 12, December 1975. (cité dans la section 6.6)

[Pfister86] G. F. Pfister et al. *An Introduction to the IBM Research Parallel Processor Prototype (RP3).* Technical Report, IBM T.J. atson Research Center Yorktown Heights, NY, 1986. (cité dans les section 1.2, 1.3.2 et 1.4)

[Pount90] Dick Pountain. *Virtual Channels: The next Generation of Transputers.* BYTE, April 1990. (cité dans les sections 1.3.1, 6.4 et 7.3.2)

[Pritch87] David J. Prithard. *Mathematical Models of Distributed Computation.* OUG-7, IOS, Amsterdam Springfield, 14–16, September 1987. (cité dans la section 3.2)

[Quill80] J.M. McQuillan. *The new routing algorithm for the ARPANET.* IEEE Transactions on Communications, Vol. COM-28, $N°$ 5, May 1980. 711-719. (cité dans la section 6.2)

[QuiWa77] John M. McQuillan and David C. Walden. *The ARPA Network Design Decisions.* Computer Networks 1 (1977) 243-289. (cité dans la section 6.2)

[Rash87] Richard Rashid et al. *Machine-Independent Virtual Memory Management for Paged Uniprocessor and Multiprocessor Architectures.* Technical report CMU-CS-87-140, Carnegie Mellon University, July 1987. (cité dans la section 5.4)

[RasRob81] R. F. Rashid and G. G. Robertson. *Accent: A Communication Oriented Network Operating System Kernel.* Proc. 8th Symp. on Operating Systems Principles, Aug. 1981, p64–75. (cité dans la section 5.4)

[ReFu87] D.A. Reed and R.M. Fujimoto. Multicomputer Networks:*Message-Based Parallel Processing.* The MIT Press, 1987. (cité dans la section 1.3.1)

[Reis79] Martin Reiser. *A Queueing Network Analysis of Computer Communication Networks with Window Flow Control.* IEEE Transactions on Communications, Vol. COM-27, $N°$ 8, August 1979. (cité dans la section 6.6)

[ReSta86] R. van Renesse and H. van Staveren. *Wide-Area Communications under Amoeba.* Report IR-117, Dept. of Mathematics and Computer Science, Vrije Universiteit, December 1986. (cité dans la section 5.2)

[Rive87] Michel Riveill. *CONKER : un modèle de répartition pour processus communicants. Application à OCCAM.* Thèse Docteur INPG, LGI-IMAG, Grenoble, Février 1987. (cité dans la section 3.5.2)

[Rosc87] A. W. Roscoe. *Routing messages through networks: an exercise in deadlock avoidance.* 7-th Occam User Group Symposium IMAG-LGI, Grenoble 1987. (cité dans la section 7.3.2)

[RosDa86] A. W. Roscoe and Naiem Dathi. *The Pursuit of Deadlock Freedom.* Technical monograph PRG-57, Oxford University, November 1986. (cité dans la section 7.1)

[RST89] R. van Renesse, H. van Staveren and A. S. Tanenbaum. *The Performance of the Amoeba Distributed Operating System.* Software-Practice and Experience, Vol. 19, March 1989. (cité dans la section 5.2)

[RTM90] Robbert van Renesse, Andrew S. Tanenbaum and Sape. J. Mullender. *The Evolution of a Distributed Operating System.* LNCS $N°$ 433, 1990. (cité dans la section 5.2)

[RTSH87] R. van Renesse, A.S. Tanenbaum, H. van Staveren and Jane Hall. *Connecting RPC-Based Distributed System Using Wide-Area Networks.* Proc. 7th International Conf. on Distr. Comp. Systems, IEEE, 1987. (cité dans la section 5.2)

[RuMu79] Harry Rudin and Heinrich Müller. *On Routing and Flow Control.* Flow Control in Computer Networks, IFIP, North-Holland Publishing Company (1979). (cité dans la section 6.4)

[SaKha85] Nicolas Santoro and Ramez Khatib. *Labelling and Implicit Routing in Networks.* The Computer Journal, Vol. 28, $N°$ 1, 1985. (cité dans les sections 6.4, 7.3.2 et 7.8)

[Seitz85] Charles L. Seitz. *The Cosmic Cube.* Communications of the ACM, Vol. 28, $N°$ 1, January 1985. (cité dans la section 1.3.1)

[Shapiro86] E. Shapiro. *Concurrent Prolog: A progress report.* Computer 19, 8 Aug. 1986, p44–58. (cité dans la section 3.5.1)

[Shyam84] R.K. Shyamasundar. *A simple Livelock-Free Algorithm for Packet Switching.* Science of Computer Programming 4 (1984) 249-256. (cité dans la section 7.3.2)

[SMM79] Howard J. Siegel, Robert J. McMillen and Philip T. Mueler Jr. *A survey of interconnection methods for reconfigurable parallel processing systems.* National Computer Conference, 1979. (cité dans la section 1.2.2)

[SS80] Mischa Schwartz and Thomas Stern. *Routing Techniques Used in Computer Communication Networks.* IEEE Transactions on Communications, Vol. COM-28, $N°$ 4, April 1980. (cité dans la section 6.2)

[Stern79] Thomas E. Stern. *Approximations of Queue Dynamics and Their Application to Adaptive Routing in Computer Communication networks.* IEEE Transactions on Communications, Vol. COM-27, $N°$ 9, September 1979. (cité dans la section 6.4)

[Sun77] Carl A. Sunshine. *Interconnection of computer networks.* Computer Networks 1 (1977) 175-195. (cité dans la section 6.2)

[TalMun90] E-G. Talbi et T. Muntean. *Placement statique de processus sur une architecture parallèle.* Rapport de recherche RR 833-I, LGI-IMAG, Grenoble, Novembre 1990. (cité dans la section 3.4)

[TanRen85] A. S. Tanenbaum and R. Van Renesse. *Distributed Operating Systems.* ACM Computing Surveys, Vol. 17, $N°$ 4, December 1985. (cité dans la section 5.1)

[Tanen89] A.S. Tanenbaum et al. *Experiences with the Amoeba Distributed Operating System.* Report IR-194, Dept. of Mathematics and Computer Science, Vrije Universiteit, July 1989. (cité dans la section 5.2)

[TeRa87] Avadis Tevanian, Jr. and Richard F. Rashid. *MACH: A Basis for Future UNIX Development.* Technical Report CMU-CS-87-139, Carnegie Mellon University, June 1987. (cité dans les sections 3.4.2 et 5.4)

[Teva87t] Avadis Tevanian, Jr. et al. *Mach Threads and the Unix Kernel: The Battle for Control.* Technical Report CMU-CS-87-149, Carnegie Mellon University, 1987. (cité dans la section 5.4)

[Teva87m] Avadis Tevanian, Jr. et al. *A Unix Interface for Shared Memory and Memory Mapped Files Under Mach.* Computer Science Department, Carnegie Mellon University, July 1987. (cité dans la section 5.4)

[Tilb89] André M. van Tilborg. *Panel on Future Directions In Parallel Computer Architecture.* Computer Architecture News. Vol. 17, $N°$ 4, June 1989. (cité dans les sections 1.1 , 1.4 et 2.7)

[Tricot84] C. Tricot. *Un modèle pour la diffusion.* Rapport DEA, LGI-IMAG, Grenoble, September 1984. (cité dans la section 8.3)

[TRM86] A.S. Tanenbaum, S.J. Mullender and R. van Renesse. *Using Sparse Capabilities in a Distributed Operating System.* Proc. 6th International Conf. on Distrib. Com. Sys., IEEE, 1986. (cité dans la section 5.2)

[ToUll81] Sam Toueg and Jeffrey D. Ullman. *Deadlock-Free Packet Switching Networks.* SIAM J. COMPUT. Vol. 10, $N°$ 3, August 1981. (cité dans la section 7.3.2)

[Tour89] Bernard Tourancheau. *Algorithmique Parallèle pour les machines à mémoire distribuée.* Thèse Docteur INPG, TIM-3-IMAG, Grenoble, Février 1989. (cité dans les sections 1.2.1 et 1.3.1)

[TuRo88] Lewis W. Tucker and Georges G. Robertson. *Architecture and Applications of the Connection Machine.* IEEE COMPUTER, August 1988. (cité dans les sections 1.1, 1.2, 1.2.1 et 2.2)

[Waille91] Ph. Waille. (*A paraître*). Thèse Docteur INPG, LGI-IMAG, Grenoble, (1991). (cité dans les sections 2.6, 6.7 et 9.1)

[Wilcke89] W.W. Wilcke et al. *The IBM Victor Multiprocessor Project.* Proc. of the 4th International Conference on hypercubes, April 1989. (cité dans les sections 1.3.1 et 2.5)

[Wittie81] Larry D. Wittie. *Communication Structures for Large Networks of Microcomputers.* IEEE transactions on Computers, Vol. C-30, $N°$ 4, April 1981. (cité dans la section 1.3.1)

[WM90] Ph. Waille et T. Muntean. *Introduction à l'architecture des machines supernode.* La lettre du transputer $N°$ 7 , 1990. (cité dans la présentation de la thèse et dans la section 2.6)

[WuFeng84] Chuan-lin Wu and Tse-Yun Reng. *Tutorial: Interconnection Networks for Parallel and Distributed Processing.* IEEE Computer Society Press, 1984. (cité dans les sections 1.3.1 et 2.3)

[WSP90] Wolfgang Schröder-Preikschat. *PEACE- A Distributed Operating System for High-Performance Multicomputer Systems.* Lecture Notes in Computer Science $N°$ 443. Progress in Distributed Operating Systems and Distributed System Management. edited by W. Schröder-Preikschat and W. Zimmer , 1990. (cité dans les sections 3.4.2 et 5.5)

[Young87] Michael Young et al. *The Duality of Memory and Communication in the Implementation of a Multiprocessor Operating System.* ACM SIGOPS, 1987. (cité dans la section 1.2.2 et dans la figure 1.1 à la page 32)

Résumé

Cette thèse aborde un ensemble de problèmes liés à la conception et à la mise en œuvre d'un noyau de communication faisant partie de Parx, un noyau de système d'exploitation pour machines multi-processeurs sans memoire commune, développé dans le cadre du projet de recherche européen ESPRIT "Supernode".

Le noyau realise une machine virtuelle, vis-à-vis des communications, dans laquelle l'ensemble de processeurs est complètement connecté indépendamment de la topologie du réseau d'interconnexion sous-jacent. La machine virtuelle offre une interface qui facilite l'exploitation correcte du haut degré de parallélisme physique des machines visées.

Après un état de l'art des architectures d'ordinateurs massivement parallèles, il est proposé un modèle de processus et une structure de noyau de système parallèle. Le modèle est basé sur un ensemble d'entités bien adaptées au contrôle de l'exécution des programmes parallèles composés de processus communicants. Ces entités, qui étendent la notion traditionnelle de processus, intègrent des concepts nouveaux visant la meilleure exploitation de l'architecture physique.

Dans le modèle de processus communicants, ceux-ci ne coopèrent que par échange de messages. Le contrôle, correct et efficace, de la communication et la synchronisation entre processus s'exécutant sur une architecture multi-processeurs sans mémoire commune est le thème central de cette thèse. Notre étude s'oriente vers la conception d'un noyau de communication, pour lequel les problèmes concernent essentiellement le routage des messages sans interblocage dans le réseau de processeurs et les protocoles de communication entre processus adéquats au modèle de programmation utilisé.

Mots clés : parallélisme, système parallèle, architectures parallèles, système réparti, processus communicants, routage des messages, interblocage.