PARX : architecture de noyau de système d'exploitation parallèle

Yves Bertrand Langue Tsobgny

THESE

présentée par

Yves Bertrand LANGUE TSOBGNY

pour obtenir le titre de **DOCTEUR**

DE L'INSTITUT NATIONAL POLYTECHNIQUE DE GRENOBLE

(Arrêté ministériel du 23 Novembre 1988)

Spécialité : **Informatique**

PARX : Architecture de Noyau de Système d'Exploitation Parallèle

Date de soutenance : 13 Décembre 1991

Composition du jury :

Président :	Jacques	MOSSIERE
Rapporteurs :	Claude	BETOURNE
	André	SCHIPER
Examinateurs :	Jacques	FEBVRE
	Traian	MUNTEAN

Thèse préparée au sein du Laboratoire de Génie Informatique

à mes chers parents

Remerciements

C'est avec grand plaisir, au terme de ce travail, que je remercie ceux grâce à qui cette thèse a pu voir le jour.

Jacques Mossière, Professeur et Directeur de l'ENSIMAG, pour s'être intéressé à mes recherches et me faire l'honneur de présider mon jury de thèse.

Claude Betourné, Professeur à l'Université Paul Sabatier de Toulouse, et André Schiper, Professeur à l'Ecole Polytechnique Fédérale de Lausanne, pour avoir accepté d'être rapporteurs de cette thèse. Par leur lecture attentive de mon manuscrit et leur remarques avisées ils ont largement contribué à son amélioration.

Jacques Febvre, Directeur du Centre de Recherches de l'Open Software Foundation à Grenoble, pour avoir bien voulu être membre de mon jury et pour l'intérêt qu'il a porté aux différentes phases du projet qui est présenté ici.

Traian Muntean, Directeur de Recherches à l'INPG, pour avoir dirigé mes premiers pas dans la recherche et sû m'initier et me guider sur des chemins nouveaux. Grâce aux nombreuses discussions que nous avons eues, nous avons pu proposer une approche originale dans le domaine du parallélisme, qui promet encore de nombreuses avancées.

Je n'oublie pas mes compagnons de recherche des équipes Sympa et Flop, avec qui chaque jour un peu plus s'est façonné Parx : mon "cher ami" Néstor González Valenzuela, Ibrahima Sakho qui veille sur nous tous, Léon Mugwaneza le marathonien, mais aussi François, Ghazali, Harold, Philippe et Philippe, Pierre, Yu, Michel, Jacques et les autres.

Il me faut enfin souligner l'excellente ambiance qui règne au Laboratoire de Génie Informatique, où je me suis vraiment senti chez moi. Pour cela, j'adresse un merci collectif à tous ses membres.

Résumé

Nous présentons ici l'architecture d'un noyau de système d'exploitation pour machines parallèles : Parx[1]. Nous discutons les outils de base pour le support d'applications parallèles au niveau d'un noyau de système. Les aspects liés aux modèles de processus et de communication sont développés.

Notre démarche s'appuie d'une part sur l'étude des modèles de programmation parallèles sous-jacents aux langages, d'autre part sur les architectures de machines parallèles modernes. Cette approche sur deux fronts convergents nous permet de prendre en compte à la fois les progrès dans l'expression et l'utilisation du parallélisme, et les tendances et les possibilités technologiques de construction de machines parallèles.

Le résultat principal de cette étude est la conception d'une architecture de système d'exploitation parallèle original et la réalisation d'un noyau de communication pour ce système.

Les systèmes d'exploitation ne peuvent plus prétendre offrir une gamme de services dont puissent se satisfaire toutes les applications. Ils doivent supporter un nombre croissant d'applications de types différents, et décomposent leurs fonctionnalités en un noyau qualifié de "micro", "léger" etc., et un ensemble de serveurs de haut niveau, s'occupant de gérer des fichiers, de la mémoire etc. C'est l'approche que nous adoptons ici.

Mots clés : parallélisme, systèmes d'exploitation parallèles, programmation parallèle, architectures parallèles, modèle d'exécution, modèle de processus, noyau de communication, protocoles de communication.

[1]Ce travail a été partiellement financé par le programme ESPRIT sous les contrats P1085 (Supernode I) et P2528 (Supernode II).

Abstract

This thesis presents the architecture of the kernel of an operating system for parallel machines. We discuss basic paradigms for supporting parallel applications at the level of the kernel of an operating system. Emphasis is given to issues related to process and communication models.

Two studies are at the basis of our reasoning: parallel programming models as defined in programming langages, and architectures of modern parallel machines. This two-pronged approach allows us to simultaneously take into account progress in expressing and actually using parallelism, and tendencies and technical improvements in parallel machines technology.

The main result of this work is the design of the architecture of an original parallel operating system, Parx, and the design and implementation of a communication kernel for that system.

Operating systems can no longer pretend to offer a range of services suitable for all types of applications. As they support a growing spectrum of applications, their functionnalities now tend to be split into a kernel called "micro", "light" or otherwise, and a set of high level servers, generally servicing files, memory etc. This is the approach which is pursued here.

Key words: parallelism, parallel operating systems, parallel programming, parallel architectures, execution model, process model, communication kernel, communication protocols.

Chapitre 1

Introduction

1.1 Objet de la thèse, apport personnel

L'objectif de cette étude est la conception d'un noyau de système d'exploitation adapté aux applications et aux machines parallèles. Pour cela, nous examinerons d'une part les modèles de programmation parallèle définis par les langages, et d'autre part les architectures de machines parallèles modernes. Cette approche sur deux fronts convergents nous permet de prendre en compte à la fois les progrès dans l'expression et l'utilisation du parallélisme, et les tendances et les possibilités technologiques de construction de machines parallèles.

Nous plaçons dans le noyau d'un système les fonctionnalités de base suivantes : un modèle de processus et de communication, pour lesquels nous proposons une approche originale. Nous n'avons pas développé d'autres aspects importants comme la protection ou la gestion mémoire, nous référant pour cela à des travaux existants [Accet86, AGHR89,Mullen90]. Enfin, nous n'abordons le domaine des services systèmes que pour introduire le support offert dans le noyau à des serveurs systèmes, notamment pour la gestion de fichiers ou d'objets.

Le résultat principal de cette étude est la conception d'une architecture de système d'exploitation parallèle original, Parx, et la conception et la réalisation d'un noyau de communication pour ce système.

Ce travail s'inscrit dans le cadre des recherches des équipes Sympa et Flop, et n'est qu'une facette d'une réalisation collective ambitieuse : le projet Parx. Il s'agit de l'étude et de la conception d'un noyau de système et d'applications à haut degré de parallélisme. Les aspects importants non développés ici, notamment la conception d'architectures de machines parallèles, les problèmes du routage correct dans de telles machines, de l'environnement de programmation, en particulier l'allocation, la définition et la compilation d'extensions de langages parallèles (C ou Prolog), la définition de nouveaux formalismes pour le parallélisme, sont autant de thèmes traités dans le groupe de tra-

vail.

Notre apport personnel consiste en la définition de l'architecture générale du système Parx, notamment des modèles de processus et de communication. Au cours de cette étude, nous avons réalisé un noyau de communication qui a fait l'objet d'un transfert industriel. Le modèle de processus a été partiellement mis en œuvre par une bibliothèque de processus légers, les threads. Les autres aspects de Parx ont été développés par d'autres membres de l'équipe pour les aspects reconfiguration, routage correct, gestion des objets entre autres, et par nos partenaires du projet Esprit Supernode II pour le reste du modèle de processus et les sous-systèmes.

La multiplication des architectures comportant plusieurs processeurs, spécialisés ou non, entraîne une certaine confusion dans la classification des systèmes informatiques. Il est donc indispensable de préciser notre acceptation de quelques-uns de ces termes controversés [BSTan89].

Un **système multi-processeur** est un ensemble de plusieurs processeurs non spécialisés partageant une mémoire ou une hiérarchie de mémoires communes. Le système peut aussi contenir des processeurs spécialisés.

Un **système réparti** est un réseau de machines ne partageant pas de mémoire, et communiquant, par échanges de messages, au moyen d'un réseau local. Chaque machine est en général un mini-ordinateur, un poste de travail ou une machine spécialisée.

Un **système parallèle** est un ensemble de "nœuds" ne partageant pas de mémoire et communiquant par échanges de messages. Le rapport entre la durée moyenne d'une séquence de traitements et la durée moyenne d'une communication, que nous définissons comme le **grain de communication**, caractérise les applications et les systèmes parallèles. La durée des communications est ici significative par rapport à la durée des traitements.

Ce rapport aborde essentiellement les systèmes parallèles. Les systèmes distribués présentent de nombreuses similitudes avec les systèmes parallèles, et notamment les mécanismes des systèmes d'exploitation mono-processeurs sur lesquels nous ne reviendrons pas. Nous adopterons une terminologie classique, inspirée de [CORNA81,Krako88].

Chacun des nœuds d'une machine parallèle comporte un ou plusieurs processeurs, spécialisés ou non. Le terme anglais "multicomputer" est plus approprié pour décrire la spécificité des processeurs formant les nœuds de ces réseaux. Chaque nœud est en fait une machine autonome. Elle peut comprendre un processeur central possédant une mémoire privée, des processeurs de communication fonctionnant parallèlement au processeur central, et des ports de communication physique.

Les systèmes répartis diffèrent des systèmes parallèles par l'importance du couplage entre les machines ou entre les nœuds. Dans les premiers, les machines sont relativement indépendantes, chacune pouvant démarrer, fonctionner et s'arrêter de façon autonome ; leur réseau d'interconnexion est lent et pas toujours fiable. Par contre dans les seconds, les nœuds sont plus fortement couplés. Il démarrent, coopèrent et s'arrêtent généralement simultanément. Le réseau d'interconnexion est rapide et fiable. Les termes "lent" et "rapide" sont tout à fait relatifs ; ils font ici référence au rapport entre la durée de traitement du processeur central et la durée de communication des dispositifs physiques pour la moyenne des applications. Ce rapport est le grain de communication matériel. Les réseaux d'ordinateurs font généralement abstraction des caractéristiques topologiques du réseau. Au contraire, celles-ci sont cruciales pour les machines parallèles.

La visibilité du parallélisme dans les systèmes informatiques a évolué au fur et à mesure qu'il était mieux maîtrisé. Dans les années 1960, la recherche de puissances de calcul toujours supérieures mène à l'introduction de processeurs d'entrées-sorties, puis de co-processeurs arithmétiques ou vectoriels, et plus tard de plusieurs processeurs centraux fonctionnant en parallèle (Illiac IV [Hord82] en 1967). Les systèmes d'exploitation dissimulent alors scrupuleusement aux utilisateurs l'existence de multiples processeurs, afin de leur présenter une machine virtuelle de haut niveau dissimulant la complexité du matériel. La commande de plusieurs périphériques asynchrones oblige les programmes systèmes à mener plusieurs activités en pseudo-parallélisme. Les programmes systèmes sont alors des programmes pseudo-parallèles dont les différentes activités sont en compétition pour l'accès au processeur. L'introduction de plusieurs processeurs généraux (multi-processeurs) permet ensuite aux systèmes de se paralléliser afin d'offrir un meilleur service aux programmes application. Enfin, l'apparition des machines parallèles donne aux utilisateurs la possibilité de bénéficier du parallélisme pour l'écriture de leurs applications.

1.2 Pourquoi des machines parallèles ?

La nature semble une source inépuisable de problèmes dont la résolution, par des modèles réalistes, demande des puissances de calcul dépassant largement celles des ordinateurs les plus performants du moment. La physique nucléaire et les prévisions météorologiques en sont des exemples. Dans le passé, de nouveaux ordres de puissance de calcul étaient régulièrement franchis par une meilleure intégration d'opérateurs de traitement en silicium. Cette approche, bien que profitable et encore prometteuse, est limitée par des contraintes phy-

siques et technologiques d'intégration d'une part, économiques de coût de construction d'autre part.

Cette limitation a renforcé le développement de modèles d'architectures de machines différents de celui de Von Neuman, et principalement l'usage de plusieurs processeurs fonctionnant en parallèle. Ce furent tout d'abord simplement des unités périphériques, puis des processeurs centraux. Simultanément, un effort considérable est accompli pour les langages et les environnements de programmation parallèles. Leur puissance d'expression est de plus en plus forte, et ils permettent la manipulation de notions que l'on rencontre dans les problèmes réels : parallélisme, coordination temporelle, etc.

La pression des multiples applications possibles, jointe aux progrès technologiques et à une meilleure connaissance des systèmes utilisables sur ces machines permettent d'observer une évolution rapide de l'usage du parallélisme dans les systèmes informatiques. On peut distinguer trois types d'architectures pour les machines parallèles, et trois modes d'exploitation du parallélisme.

Un premier type d'architecture est illustré par la *Connection Machine* [TuRo88] et le *Massively Parallel Processor* [Batcher80], où un très grand nombre (plusieurs milliers) de processeurs très simples (parfois des processeurs un-bit) travaillant en parallèle permettent d'obtenir de fortes puissances de calcul.

Prenant le contrepied de ce premier type, les super-calculateurs[1], tels que les machines de Cray Research, utilisent un petit nombre (deux à quatre, en excluant les modèles Cray III et Cray IV dont les sorties commerciales sont attendues pour 1991 et 1995) de processeurs de très forte puissance (de l'ordre du Gigaflop).

Un dernier type couvre le spectre allant de l'un à l'autre des extrêmes déjà cités : les réseaux de processeurs. Des processeurs, pouvant être regroupés en nœuds qui sont parfois de véritables machines, communiquent, à travers un réseau d'interconnexion, avec d'autres processeurs (figure 1.1 a) ou avec de la mémoire (figure 1.1 b).

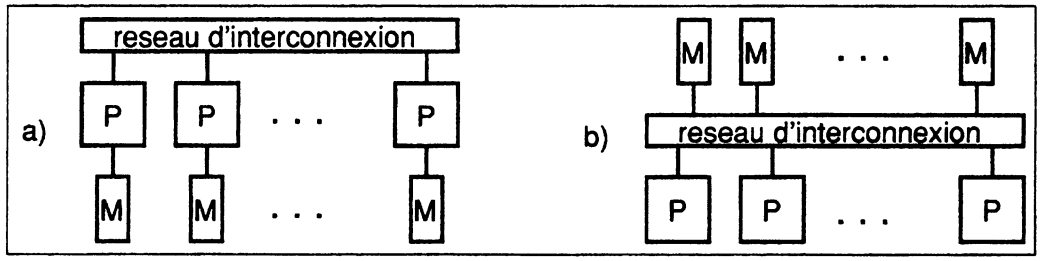


Figure 1.1: Schémas d'interconnexion des réseaux de processeurs

Les réseaux de processeurs de taille importante (plus d'une vingtaine de processeurs) ne partagent en général pas la même mémoire, la

[1] ordinateurs dont la puissance dépasse 500 Mflops [Hwang87]

technologie actuelle ne permettant pas l'accès simultané de dizaines ou centaines de processeurs à la même mémoire avec des temps de réponse acceptables. Nous nous intéressons principalement à ce dernier type d'architecture, bien que les paradigmes proposés soient souvent valables pour les deux autres.

Les machines parallèles sont l'objet d'un intérêt de plus en plus marqué. Avec la multiplication d'architectures différentes et leur banalisation dans la communauté scientifique, la recherche a fait des progrès importants ces dernières années, bien que l'on commence seulement à prendre conscience de l'étendue des applications du parallélisme.

Cependant, malgré la multiplicité des machines parallèles, il n'y a encore que très peu de systèmes d'exploitation parallèles. Plusieurs projets sont en cours sur ce sujet, dont Parx, en développement à l'IMAG, qui fait l'objet de ce rapport.

1.3 Organisation de la suite du rapport

Le domaine des machines parallèles est réputé, à juste titre, difficile. L'absence de réalisation s'imposant comme référence ou comme standard vient à l'appui de cette constatation. Au contraire, de nombreuses recherches ont été menées dans des domaines connexes, les systèmes multi-processeurs et les systèmes répartis [Pfister86, Lazow81,Bal87]. Plusieurs architectures de machines, systèmes d'exploitation, environnements de programmation, bibliothèques, langages, compilateurs et applications sont disponibles pour ces derniers. Cependant, si les machines parallèles sont désormais nombreuses, le support logiciel, comme nous le verrons au chapitre 3, fait encore cruellement défaut. Les problèmes à résoudre sont, il est vrai, complexes, et il est difficile de les aborder simultanément de front.

Cette étude adopte une approche pragmatique aux problèmes des mécanismes de base (gestion de processus et communication) pour l'utilisation des systèmes parallèles, tout en restant en dehors des contextes habituels trop liés à une application ou à une machine particulières. Son champ d'application expérimental est la famille des machines Supernode [HJMW87,SnI89], mais nous le généralisons à des machines parallèles connectées en réseau maillé statique ou reconfigurable quelconque. Nous nous intéressons principalement au cas général où les nœuds du réseau ne partagent pas de mémoire principale.

Ce rapport est divisé en deux parties. La première, regroupant les chapitres 2, 3 et 4, vise, à travers l'étude des caractéristiques des langages et des architectures de machines parallèles, à dégager le type

et les spécificités d'un système d'exploitation qui leur soit adapté. Le principe est le même pour les deux premiers chapitres de cette partie : mettre en évidence les modèles de parallélisme et de communication, afin d'en déduire le support correspondant à offrir au niveau d'un noyau de système. Le second chapitre présente une étude de l'expression du parallélisme et de la communication dans les langages de programmation. Elle donne des indications précises sur le support système "idéal" pour les applications parallèles. Nous étudions les modes d'expression du parallélisme dans les langages de programmation, puis la communication et la synchronisation entre entités parallèles. Dans le troisième chapitre, nous étudions les caractéristiques des architectures parallèles visées, principalement les machines MIMD sans mémoire commune. Nous nous intéressons au type d'exploitation du parallélisme et à la mise en œuvre de la communication à travers quelques architectures de machines parallèles. Nous dégageons un ensemble de propriétés intéressantes pour une architecture parallèle. Cette première partie se termine avec l'étude de la machine Supernode, qui est l'une des machines cibles de notre travail.

La seconde partie, du chapitre 5 au chapitre 7, présente l'architecture du noyau de système Parx. A partir des résultats de la première partie, nous proposons l'architecture du noyau de système Parx. Le chapitre 5 introduit et justifie les concepts de base. Il précise les objectifs du projet Parx et l'approche adoptée dans le contexte de la recherche actuelle. Il décrit l'architecture générale du système. Le chapitre 6 détaille le modèle de processus de Parx. Au chapitre 7 est présenté le noyau de communication de Parx. Son principe est décrit, ainsi que les protocoles qu'il comprend et leur fonctionnement. Dans chacun de ces deux derniers chapitres, nous présentons le problème traité, l'approche et la solution adoptée dans Parx, ainsi que l'interface de manipulation. Enfin, le chapitre 9 présente brièvement quelques autres systèmes parallèles et situe leurs différents choix par rapport à ceux adoptés dans Parx.

Nous concluons enfin par le bilan de ce travail et en projetant des perspectives pour le futur.

Partie I

Support Système pour Modèles de Programmation et Exploitation d'Architectures Parallèles

Chapitre 2

Modèles de programmation dans les langages parallèles

Notre objectif dans ce chapitre est de mettre en évidence les modèles de programmation parallèle qui apparaissent à travers les langages de programmation. Cette démarche est importante parce qu'elle permet au concepteur de systèmes d'exploitation de mieux répondre aux besoins ainsi exprimés. Pour cela, nous étudions les modèles de programmation des langages, notamment parallèles, afin d'en déduire le support système qu'ils requièrent. La définition dans un langage d'entités et d'opérations conduit à une machine abstraite. L'étude de cette machine nous permet de mieux concevoir le support système apte à la mettre en œuvre.

Une première constatation est qu'il y a des opinions très différentes quant aux fonctionnalités à placer dans le système d'exploitation, dans l'environnement langage et le compilateur, et celles dont il faut laisser le soin de la programmation à l'utilisateur.

La répartition des fonctions, si elle est relativement claire pour la programmation classique sur les machines séquentielles, est à créer pour la programmation parallèle. En effet, l'exécution d'une application parallèle met en œuvre, de façon explicite ou implicite pour le programmeur, et en fonction du langage, la manipulation d'entités actives, l'ordonnancement de leur exécution, l'attribution de priorités, le contrôle du placement sur des processeurs spécifiques ; toutes tâches traditionnellement réservées au système d'exploitation. Certains environnements de programmation, comme occam Toolset [Inmos88], intègrent toutes ces fonctionnalités de façon plus ou moins réussie, et se passent complètement de système d'exploitation.

A l'étude des modèles de programmation, il apparaît que chacun demande l'application d'un ensemble de politiques pour la gestion des entités qu'il définit. Les décisions demandent une connaissance que ne peut en général avoir le système, même en examinant tout le cours de l'exécution passée du programme.

Le système d'exploitation, pour permettre de supporter des langages différents utilisés simultanément par des usagers différents, doit être capable de fournir des modèles d'entités compatibles avec ceux des langages, et des opérations de base à partir desquelles des politiques différentes puissent être élaborées, sans demander un effort "trop important" (appréciation subjective) d'adaptation.

La première section discute brièvement les modèles de programmation parallèle. La seconde section s'intéresse aux différents modes d'expression du parallélisme et du pseudo-parallélisme, et au contrôle associé. La troisième étudie la collaboration, la communication et la synchronisation entre entités parallèles.

2.1 Les modèles de programmation

La recherche de modèles de programmation parallèle est un domaine intéressant, et plusieurs propositions ont été avancées. Il n'y a cependant pas encore de modèle reconnu de tous. Plusieurs écoles se distinguent dans la recherche des outils de base de la programmation parallèle, autour desquels sont bâtis tous les programmes parallèles. Cette recherche est importante parce qu'elle permet de définir une plateforme capable d'accueillir de nombreux programmes parallèles, quelque soit le langage dans lequel ils sont exprimés.

Nous nous intéressons dans cette section aux squelettes de programmation sous-jacents de très bas niveau que doit supporter le modèle d'exécution offert par le système d'exploitation. Nous citons deux propositions et analysons leur démarche. Pritchard [Pritch87] et Muntean [Munt88] définissent trois modèles de programmation : les parallélismes réplicatif, géométrique et algorithmique. Carriero et Gelernter [CaGel89b] proposent trois classes conceptuelles de parallélisme : il s'agit des parallélisme résultat, spécialisé et agenda. Ils correspondent respectivement à trois techniques qui sont les structures de données actives, l'échange de messages et l'utilisation de structures de données distribuées.

Dans ces deux propositions, les démarches sont semblables : il s'agit de combiner les différentes façons d'appliquer des traitements à des données. Deux ensembles, l'un de traitements, l'autre de données, sont chacun décomposés. Les sous-ensembles obtenus regroupent des éléments suivant une relation de dépendance particulière. Cette relation peut être de différentes natures : dépendance temporelle, spatiale, logique etc. Par exemple, des traitements exécutables en parallèle seront temporellement indépendants. On peut aussi exploiter la structure géométrique d'un problème, par exemple en traitement d'images, pour le décomposer.

Un modèle de programmation correspond au choix des décompositions

des traitements et des données et de leurs interactions. Devant le nombre élevé de combinaisons possibles, il apparaît bien difficile de donner une méthodologie de programmation universelle ou même s'en approchant. Il n'existe pas de modèle encapsulant les raisonnements de tous les programmeurs.

De ces différentes approches de la programmation parallèle, il découle la nécessité de différentes techniques de parallélisme répondant à la vaste palette des besoins des applications, et ce pour une même architecture. Il faut donc considérer chaque application parallèle comme un tout auquel une politique globale cohérente de gestion doit être appliquée. Cette politique peut largement différer d'application en application. Deux champs de recherche s'ouvrent alors :

- quelle décomposition adopter pour les unités de traitement, et quel grain fournir ?
- quel modèle et quelle gestion pour les objets distribués ?

Le premier axe fait l'objet du chapitre 6, le second est un sujet de recherche étudié dans une autre thèse en cours dans notre équipe.

2.2 Modes d'expression du parallélisme

A la différence des programmes séquentiels, l'exécution d'un programme parallèle met en jeu simultanément plusieurs flots d'exécution et éventuellement plusieurs environnements associés. Un flot d'exécution, lors de l'accès à un environnement extérieur, doit donc tenir compte de la possibilité d'une évolution indépendante de son propre environnement. L'ordonnancement des flots d'exécution et leur synchronisation ne peuvent se faire que sur la base d'informations spécifiques au programme. L'influence des programmes sur ces informations est plus ou moins importante suivant les langages. Elles constituent une contrainte à prendre en compte par le système d'exploitation pour l'exécution du programme.

Chaque langage regroupe un ensemble de fonctionalités et de propriétés dans une "entité". Celle-ci est couramment appelée processus ou objet. Ces entités sont le mode d'expression du parallélisme dans le langage.

2.2.1 Panorama des langages de programmation

Le développement de la programmation parallèle, avant celui de systèmes parallèles et notamment depuis les travaux de Hoare [Hoare78]

et Milner [Milner80], popularise les notions de processus et leur exécution parallèle.

Dans les langages, des primitives ou des constructions permettant d'exprimer du parallélisme explicite, ou d'extraire du parallélisme implicite sont introduites.

Les langages les plus connus sont CSP [Hoare78] qui engendra toute une famille de langages dont occam [Inmos84b] avec la communication par échanges synchrones de messages, NIL [StYem83] avec échanges asynchrones de messages, Ada [Ichbiah79] avec le rendez-vous, Distributed Processes [Brinch78] et Cedar [SZH85] avec l'appel de procédure à distance, SR [Andrew81] avec un mélange de différentes primitives de communication, Concurrent Smalltalk [YoTo86], Emerald [Black86] ou Guide [Frey91] avec une approche objet, Argus [Liskov88] avec les transactions atomiques, ParAlfl [Hudack86] ou Concurrent Prolog [Shapiro86] pour la programmation logique, et Linda [Geler85] pour les données distribuées (voir tableau 2.1[1]).

Ils proposent un ensemble de modèles de programmation que [BSTan89] ou [CaGel89b] résument dans d'excellents états de l'art. Certains se démarquent complètement des courants de programmation traditionnels, d'autres se contentent d'étendre des langages séquentiels existants. L'intérêt de cette dernière approche est de permettre aux programmeurs un abord progressif de la programmation parallèle, tout en préservant l'acquis développé à l'aide du langage de base.

2.2.2 Entités parallèles

Le parallélisme exprimé dans les langages est en général *potentiel*. Il n'est réel que si le langage définit expressément une projection des entités parallèles sur des processeurs physiques. La distribution des entités est alors physique (voir ligne "Projection physique" du tableau 2.1). Dans les autres cas, il s'agit de pseudo-parallélisme, et la distribution est alors logique.

Le parallélisme logique peut être implicite, c'est généralement le cas des expressions et des clauses, ou explicite pour les processus, objets et constructeurs. Le parallélisme implicite est généralement détecté par des opérations simultanées sur une même variable de structure complexe, ou bien des opérations indépendantes sur des variables différentes. Les variables sont alors partagées entre des entités créées par le compilateur ou le système d'exploitation. Quand il est explicite, des commandes du langage désignent les entités parallèles et commandent leur gestion.

Dans le parallélisme physique, le programmeur peut, de plus, intervenir sur la localisation et le regroupement des entités.

[1] inspiré de [BSTan89, p270]

Aspect abordé	Exemples de langages
A) Parallélisme	
1) Expression	
Processus	Ada, Concurrent C, Linda, NIL
Objets	Emerald, Actors, Guide
Constructeurs	occam
Expressions	ParAlfl, FX-87
Clauses	Concurrent PROLOG, ParAlfl
2) Projection Physique	
Statique	occam, StarMod
Dynamique	Concurrent PROLOG, ParAlfl
Migration	Emerald
B) Communication	
1) Echange de messages	
point à point	CSP, occam, NIL
Rendez-vous	Ada, Concurrent C
Appel de procédure à distance	DP, Concurrent CLU, LYNX
Un vers plusieurs	BCSP, StarMod
2) Données partagées	
Structures de données distribuées	Linda, Orca
Variables logiques partagées	Concurrent PROLOG, PARLOG
3) Non déterminisme	
Constructeur de sélection	CSP, occam, Ada, Concurrent C, SR
Clauses de Horn gardées	Concurrent PROLOG, PARLOG
C) Pannes partielles	
Détection de panne	Ada, SR
Transaction atomiques	Argus, Aeolus, Avalon
Tolérance aux pannes transparente	NIL

Table 2.1: Primitives pour le parallélisme illustrées par des langages

2.2.2.1 Entités explicites et implicites

[BSTan89] distingue les processus, les objets, les constructeurs, les expressions et les clauses comme modes d'expression du parallélisme. Ces modes sont conceptuellement différents mais peuvent être regroupés par identité de support système.

Une première catégorie comprend les modes d'expression explicites, dans lesquels des entités actives sont spécifiquement identifiées (voir figure 2.1).

Un programme peut être décomposé en processus séquentiels s'exécutant en parallèle. Chaque processus est un flot de contrôle d'instructions exécutées dans l'environnement du programme parallèle. Il faut définir :

1. la politique de gestion qui leur est appliquée, et notamment leur ordonnancement,

```
VALUE A, B:
PROC affine (CHAN OF INT valeur, CHAN OF INT resultat ) =
CHAN pipeline :
.  PAR
.    INT x:
.    SEQ -- calcul de A * x = X
.      valeur ?  x
.      pipeline !  x * A
.    INT y:
.    SEQ -- calcul de X + B
.      pipeline ?  y
.      resultat !  y + B
```

Le constructeur PAR demande l'exécution de processus en parallèle. Les constructeurs délimitent des blocs qui définissent la portée des variables. Celle-ci s'étend au bloc directement encapsulant. Ici, "x" et "y" sont chacune privée dans l'environnement de chacun des deux processus. "pipeline", "valeur" et "resultat" sont des canaux de communication partagés par les environnements des deux processus.

PLACED PAR permet d'assigner l'exécution d'un processus à un processeur physique.

```
PLACED PAR
.  PROCESSOR 1
.    affine(valeur, resultat)
.  PROCESSOR 2
.    terminal(term.in, term.out)
```

Figure 2.1: Illustration des entités explicites en occam

2. leur synchronisation lors de l'accès à l'environnement, et
3. leur utilisation des ressources du système.

Les objets encapsulent un ensemble de données et les procédures d'accès à ces données. L'interaction entre objets se fait par échanges de messages. Dans la programmation parallèle, les objets sont en général actifs ; leur exécution peut commencer et continuer sans invocation explicite par échange de messages. Dans ce cas, on observe différents flots de contrôle s'exécutant en parallèle au sein d'un même objet.

Les constructeurs parallèles d'occam permettent de demander l'exécution d'instructions en "parallèle". Chaque flot d'instructions peut à son tour contenir des constructeurs parallèles. Les constructeurs définissent en fait un treillis de séquences de processus ne partageant pas de variables. Ils équivalent donc aux processus séquentiels de CSP. Les sommets du treillis sont les constructeurs parallèles, les arcs les processus séquentiels.

Une seconde catégorie regroupe les expressions et les clauses. Le parallélisme est ici largement implicite, quelques langages permettant au programmeur d'intervenir en fournissant des indications. L'écriture des programmes elle-même nourrit le parallélisme, ce qui est une alternative au parallélisme explicite professé par la première catégorie. Il s'agit des programmations fonctionnelle et logique.

Dans ces deux modes, des expressions ou des clauses sont à évaluer, ce qui peut être fait soit séquentiellement, soit en parallèle. Nous utiliserons le seul terme "expression" pour tout le reste de cette section. On distingue la conjonction (ET parallélisme) d'expressions et leur union (OU parallélisme), illustrés par la figure 2.2.

On pose des questions à un indicateur de chemins de fer. La question

"existe-t-il un chemin de la gare <départ> à la gare <arrivée> ?"

correspond au prédicat *chemin*, pour lequel les clauses suivantes sont données :

$$chemin(S, D) : -chemin(S, X) \wedge chemin(X, D) \quad (2.1)$$
$$chemin(S, S) : -VRAI \; - \textit{clause toujours vraie} \quad (2.2)$$

Pour chercher un chemin de S à D, on peut exploiter le parallélisme comme suit :

1. évaluer les deux clauses 2.1 et 2.2 en parallèle, dès que l'une des deux évaluations termine, le résultat final de l'évaluation de la clause est connu (OU parallélisme).
2. la clause 2.1 demande la démonstration de deux prédicats. Celles-ci peuvent être menées en parallèle, jusqu'à ce que toutes les deux terminent (ET parallélisme).

Figure 2.2: Parallélisme implicite en programmation logique

Dans la conjonction, toutes les expressions doivent être évaluées, et pour le cas de la programmation logique, si l'une s'évalue à FAUX, le résultat global est connu. L'évaluation peut alors s'arrêter. Dans l'union, le résultat global est connu dès que l'une des expressions s'évalue à VRAI. Il peut cependant être nécessaire de la continuer pour explorer toutes les solutions. Dans tous les cas, le problème des effets de bord des expressions est crucial. En effet, les effets de bord résultant de l'évaluation des expressions validées doivent être propagés.

Deux cas peuvent se présenter :

- l'évaluation d'expressions doit être interrompue avant leur terminaison parce que le résultat global est connu. Il faut alors annuler les effets possibles des évaluations interrompues.
- des expressions évaluées en parallèle partagent des variables. Il peut alors y avoir des problèmes de cohérence. Plusieurs solutions sont possibles, comme par exemple de privilégier une évaluation indiquée par le programmeur.

2.2.2.2 Regroupement d'entités sur des processeurs

La définition d'entités parallèles est éventuellement suivie de leur allocation aux processeurs. Cette allocation dépend fortement des intentions du programmeur quant à l'usage du parallélisme. Il peut en user pour :

- améliorer l'efficacité du traitement de son problème ; soit globalement, c'est-à-dire en allouant de façon homogène la puissance de traitement à tout le programme, soit localement pour des parties distinguées.
- augmenter la disponibilité d'un service.
- permettre la résistance aux pannes. Cet usage est souvent lié au précédent.

A l'examen de ces différentes intentions pouvant influer sur l'allocation, nous constatons qu'il faut distinguer les entités *collaborant* à la résolution d'un problème de celles en *compétition* pour accéder à la puissance de traitement, par exemple en essayant plusieurs solutions en parallèle, la première satisfaisante étant retenue.

Ces deux types d'entités demandent des ordonnancements spécifiques. [MaPu88] étudie le comportement des programmes parallèles et identifie deux politiques d'ordonnancement. Dans l'une, les entités parallèles sont ordonnancées par regroupements de collaboration. Toutes les entités d'un groupe doivent être exécutées simultanément. Dans l'autre, les regroupements correspondent à une compétition. Une seule des entités d'un groupe peut utilement être exécutée à un instant donné.

Les problèmes de l'allocation et de l'ordonnancement sont complexes. Une première solution consiste tout simplement à abandonner à l'utilisateur la charge de les résoudre. Une autre est de les prendre en compte au niveau du système d'exploitation ou de l'environnement d'exécution du langage. La difficulté est que ces derniers n'ont pas, en général, une vue globale du problème traité, ni une compréhension de ses différentes parties, qui permettrait de n'optimiser que ce qui est nécessaire. En effet, il ne sert à rien d'optimiser une portion de programme n'intervenant que pour une très faible part dans la performance globale du programme. D'autre part, l'optimisation globale sans critères précis de sélection est le plus souvent impossible.

2.3 Communication et synchronisation entre entités parallèles

L'exécution pseudo-parallèle de programmes systèmes a conduit à la définition d'un ensemble de méthodes de communication et de synchronisation. Les outils classiques sont les moniteurs, les sémaphores ou les boîtes aux lettres entre entités, essentiellement en mémoire partagée. Diverses versions opérant en mémoire non partagée en ont été proposées.

Les différents modèles de communication et synchronisation peuvent se ranger en deux classes : l'échange de messages et les données

partagées. La synchronisation est parfois utilisée dans des mécanismes de résistance aux pannes.

2.3.1 Echange de messages

L'échange de messages nécessite un emplacement pour conserver l'information échangée ; nous l'appelerons un **réceptacle**. Selon le procédé de nommage du réceptacle, le protocole de communication utilisé, la synchronisation ou les opérations de communication, une large gamme de variations existe pour ce modèle.

Le nommage du réceptacle peut être direct si l'émetteur (resp. le récepteur) donne le nom de l'entité (ou la collection d'entités) active(s) réceptrice(s) (resp. émettrice). Il est indirect s'il désigne un objet connu de tous les participants à l'opération (port, file d'attente de messages etc.).

Le protocole de communication régit le déroulement des requêtes adressées au réceptacle. Les émissions ou les réceptions peuvent être mises en file, avoir chacune une priorité, être identifiées par des "clés", selon l'état des entités correspondantes ou encore leur nom. La communication peut être effectuée entre une collection d'émetteurs et de récepteurs, chacune des collections pouvant être réduite à l'unité.

La possibilité de pertes de messages peut aussi être prise en compte dans un protocole, et signalée aux entités en cause. Une émission peut être suivie d'une réponse du récepteur. L'émetteur, par rendez-vous ou appel de procédure par exemple, peut recevoir une réponse que le récepteur aura élaborée.

A chaque communication est aussi associé un mode de synchronisation particulier. Elle peut être complètement asynchrone si les protagonistes ne peuvent avoir, par le protocole de communication, aucune information, même fugitive, sur leurs états passés. En général, pour assurer que les messages ne se perdent pas, une synchronisation est imposée à l'émetteur ou au récepteur, garantissant alors que le message a effectivement été reçu ou sera reçu, ou bien qu'un message est disponible. Il est parfois utile de préciser une capacité pour le réceptacle. Dans ce cas, une synchronisation sur l'état de remplissage de cette capacité (réceptacle plein ou vide) peut être effectuée.

Enfin, l'opération de réception peut être explicite ou implicite. Dans le second cas, le récepteur associe différents flots d'exécution à la réception de messages. Ces flots seront exécutés en dehors du flot principal.

Les réceptacles peuvent souvent être combinés dans des opérations de communication, permettant ainsi de désigner un ensemble de réceptacles, de mettre sur pied une politique de gestion des réceptacles (non déterminisme, priorités), et de synchronisation propres à un protagoniste. Ceci est la base d'un mécanisme d'abstraction permettant

à l'un des protagonistes de construire des objets de communication privés. Nommage du réceptacle et protocole de communication sont illustrés par la figure 2.3.

nommage du réceptacle	- direct : émetteur/récepteur CSP : P Q Q ! 1 P ? x - indirect : objet de communication, multiple ou non occam : P Q c ! 1 c ? x
protocole de communication	- sélection de l'information du réceptacle (clés, priorités, état du réceptacle ou des correspondants etc.) ex : ALT occam : sélection non déterministe parmi les émetteurs prêts - synchronisation lors des accès (occam : rendez-vous) - opérations sur l'information (ex : appel de procédure à distance)

Figure 2.3: Illustration des protocoles d'échanges de messages

Une très large variété de protocoles de communication existe, et des travaux [Riveill87] ont été menés pour déterminer un système générateur pour cette variété. L'idée, qui est aussi mise en application dans les *"streams"* d'Unix, est de construire des protocoles à la demande sur la base de protocoles déjà existants. Une base de "connecteurs" primitifs est définie, et des opérations de construction sont fournies. On observe de nombreuses équivalences entre protocoles.

2.3.2 Données partagées

La possibilité pour une entité d'accéder à une donnée définit l'ensemble des entités pouvant utiliser cette donnée comme moyen de communication. Les techniques d'implantation de la mémoire partagée conduisent à deux grandes familles de données partagées.

Dans la première, les données sont universellement manipulables. Un ensemble d'opérations, exclusives ou non, sont définies pour accéder à chaque donnée partagée. A nouveau, 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. Linda [CaGel89a] est représentatif de cette famille (voir figure 2.4).

Dans la seconde, 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. Un mécanisme permet d'enrichir cette technique plutôt rigide : il est possible d'affecter à une variable partagée une structure comprenant des valeurs et des variables. Ces dernières

Linda propose la communication et la synchronisation par échanges de *n-uplets* contenus dans une mémoire associative commune : l'espace des *tuples*. Le langage adopte une approche originale pour la programmation parallèle :

- 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 valués et d'autres 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.

Les n-uplets peuvent être actifs, on parle alors de processus, ou passifs, ce sont alors des données. Les processus s'exécutent en pseudo ou en vrai parallélisme. Pendant qu'ils sont actifs, les processus sont dans l'espace des tuples. Ils redeviennent des données ordinaires à leur terminaison.

Les opérations possibles sur les tuples sont :
- **out(tuple)** ajoute le n-uplet à l'espace des tuples. Non bloquant.
- **in(tuple)** recherche un tuple ayant la même valeur pour les éléments valués du n-uplet. Il est retiré de l'espace des tuples, synchrone.
- **rd(tuple)** a le même effet que "in", mais le tuple n'est pas retiré de l'espace.
- **eval(tuple)** a le même effet que "out", avec en plus la création d'un nouveau processus pour évaluer le n-uplet. Non bloquant.

Les applications principales du modèle de programmation Linda sont celles à données distribuées entre plusieurs processus. Chaque processus peut directement accéder à la donnée qu'il recherche. L'accès associatif permet de sélectionner la portion de la donnée intéressant le processus. Soit par exemple un tableau T distribué, indexé par des indices i : $(T(i, ?A))$ permet d'accéder à ses différents éléments.

Figure 2.4: Le langage Linda

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 cette famille.

2.4 Conclusion

En conclusion de ce tour d'horizon du parallélisme tel qu'il est exprimé dans les langages de programmation, nous pouvons faire un bilan des supports systèmes nécessaires.

L'expression du parallélisme passe par une entité représentant un flot d'exécution dans un environnement donné. Plusieurs flots peuvent s'exécuter en parallélisme ou pseudo-parallélisme, et les politiques d'ordonnancement doivent être adaptées aux environnements particuliers. Certains flots sont en compétition, d'autres en collaboration. A chaque politique on peut associer une entité administrative, correspondant par exemple à un ensemble de processus en compétition, et plusieurs de ces entités collaborant.

Il est d'autre part nécessaire d'encapsuler des données et des procédures (comportements) dans des objets qui peuvent être distribués sur plusieurs processeurs.

Un problème important est la détermination du domaine d'action des entités actives, afin de maîtriser les effets de bord. Ceci permet de définir des opérations affectant tout un environnement, comme la migration ou encore la jonction des résultats d'une évaluation parallèle.

Il faut enfin pouvoir fournir à l'utilisateur des outils pour manipuler le parallélisme effectif.

Différents modes de collaboration entre entités parallèles sont proposés. Ils sont fortement déterminés par les applications et il est préférable que chacune puisse construire son propre protocole à partir d'une base variée, proposant l'échange de messages et les données partagées. Les protocoles peuvent être synchrones ou asynchrones, de plusieurs vers un, ou de un vers plusieurs.

Nous pouvons résumer tout ceci en trois propositions :

1. gestion d'un programme parallèle, en particulier la distribution des procédures et des données, comme un ensemble cohérent aux besoins spécifiques,
2. nécessité d'offrir plusieurs grains de communication et de parallélisme et de pseudo-parallélisme,
3. nécessité de la coexistance de multiples protocoles.

Ces trois propositions sont mises en œuvre dans Parx, décrit dans la deuxième partie de ce rapport.

Le chapitre suivant étudie les caractéristiques des architectures de machines parallèles, afin d'en retirer un support système pour le parallélisme.

Chapitre 3

Les architectures des machines parallèles

Ce chapitre étudie les architectures des machines parallèles. Le premier objectif est de reconnaître les caractéristiques architecturales qui conduisent à faciliter des utilisations particulières. Chaque point ainsi examiné est illustré par une mise en œuvre concrète à travers une machine.

Nous avons volontairement omis les machines vectorielles, la vectorisation n'étant pas propre aux machines parallèles et étant largement étudiée par ailleurs [AlGot89,Schon87].

Le choix d'une architecture de machine permet de fixer plusieurs paramètres délimitant les possibilités de la machine. Ce choix détermine aussi le type de système d'exploitation adapté à la machine. De plus en plus de recherches sont menées sur le sujet du choix d'une architecture de machine pour des applications d'un type donné et réciproquement [Ferra90,NBW88]. [ReFu87] effectue une étude approfondie des réseaux d'interconnexion des architectures multi-processeurs.

La première section aborde les modèles d'exploitation du parallélisme, qui correspondent à des classes d'architectures différentes ; le parallélisme des données correspondant à la classe SIMD, et le parallélisme des traitements qui caractérise les machines MIMD.

Les modèles de communication sont traités dans la section suivante. Les communications par mémoire partagée et par échanges de messages sont étudiées. Différentes structures de réseaux d'interconnexion sont décrites.

Enfin, la dernière section examine les fonctionnalités offertes par les différents processeurs des machines parallèles : traitement, communication, calcul flottant en particulier. Nous étudions l'équilibrage de leurs puissances respectives. Dans tout le chapitre, les machines présentées ne le sont que dans le but d'illustrer un point particulier.

Nous avons fait le choix de ne développer qu'un seul aspect pour chacune des machines. La présentation complète de chacune des machines est en dehors de notre propos. De ce fait, leur présentation est nécessairement incomplète.

3.1 Modèles d'exploitation du parallélisme

La recherche d'une solution parallèle à un problème passe par la détermination de ce que l'on peut paralléliser. Une solution algorithmique parallèle s'exprime en général comme un ensemble connu de traitements à appliquer à un ensemble de données, pouvant être produites lors de phases de traitement ou acquises d'un support quelconque.

Nous pouvons distinguer deux classes de parallélismes. Il s'agit du parallélisme des opérations appliquées à des données différentes ("data parallelism") et du parallélisme entre différents flots de contrôle (couramment appelés "threads"). Le premier se caractérise par l'exécution en parallèle de plusieurs (quelques milliers) opérations sur des données différentes, et correspond à une architecture SIMD. Le second correspond à une architecture de machine MIMD. Nous ne rappelerons pas les différentes classifications des machines parallèles. Nous renvoyons à [Fly72] pour la celle de Flynn, [Kuck78] pour celle de Kuck, [TBH85] pour celle de Treleaven et à [Young87] pour celle de Young. Notons tout de même que, au gré des points étudiés par les auteurs, les aspects soulignés par les classifications diffèrent. Plusieurs machines mélangent les techniques faisant office de critères de classification. Notre propos couvre principalement les machines MIMD sans mémoire commune.

3.1.1 Parallélisme des données

3.1.1.1 Principe

L'ensemble des données est découpé à une granularité plus ou moins fine en données élémentaires. Un processeur virtuel est chargé d'exécuter tous les traitements que doit subir une donnée élémentaire. Une relation de dépendance permet de sérialiser et de synchroniser les traitements à accomplir sur chaque donnée élémentaire. Cette relation est un ordre partiel sur les traitements et mène à plusieurs exécutions parallèles possibles. Un ordre d'exécution est déterminé par les outils de transformation, et une séquence d'instructions et un ensemble de flots de données sont produits.

Cette décomposition permet de mettre en évidence les propriétés "spatiales" du problème et convient surtout à ceux ayant une bonne

localité spatiale par rapport à la relation de dépendance. Le modèle de programmation dit "parallélisme géométrique", étudié en section 2.1, correspond à ce modèle de parallélisme. La mise en œuvre du parallélisme passe par une décomposition des données à manipuler.

La Connection Machine [TuRo88] est tout à fait typique de cette approche, et nous allons y examiner la mise en œuvre de ce parallélisme.

La Connection Machine est née de recherches menées au Massachussett Institute of Technology. Celles-ci se sont concrétisées avec la construction des premiers modèles par la compagnie Thinking Machines en 1985. Une seconde version, la CM2, voit le jour quelques années plus tard.

3.1.1.2 Mise en œuvre

Le matériel

Un séquenceur alimente en instructions un ensemble de processeurs de traitement. Les processeurs sont regroupés par nœuds, chacun contenant 16 processeurs, une mémoire et un routeur (voir figure 3.1[1]). Chaque processeur opère sur des données contenues dans la mémoire.

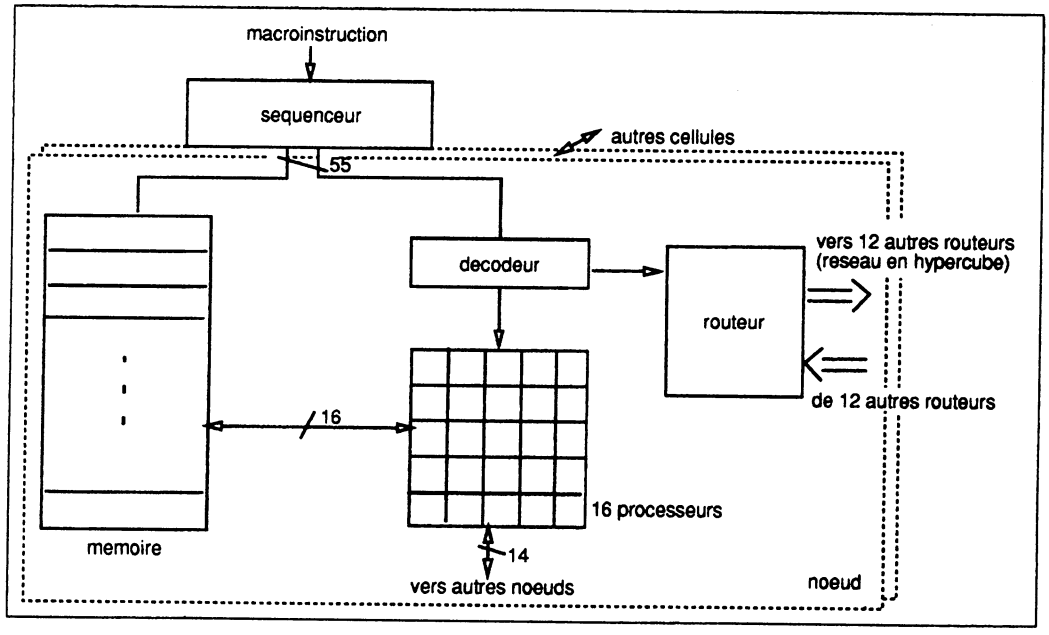


Figure 3.1: Un nœud de la Connection Machine

Chaque processeur comporte un registre d'état. Chaque opération est une fonction booléenne sur 3 bits (2 lus en mémoire, 1 du registre

[1]inspirée de [AlGot89]

d'état) et délivrant un résultat sur 2 bits (l'un est écrit en mémoire, l'autre dans le registre d'état).

La machine possède un à quatre séquenceurs qui alimentent les nœuds en instructions. Chaque nœud exécute une instruction sur une partie des données. L'accumulation de milliers de processeurs opérant en parallèle permet de délivrer de fortes puissances. Cette puissance va de 5 Gflops à 20 Gflops si le calcul ne demande pas d'échange de données entre processeurs. Elle est de l'ordre de 2,5 à 4 Mips pour de l'arithmétique entière sur 8 à 32 bits.

La CM se veut une machine générale, aux applications variées en géophysique, en intelligence artificielle ou en traitement d'images. Elle n'est pas la seule machine de référence dans la classe SIMD. Illiac IV [Hord82] ou GF11 [BDW87] peuvent être cités parmi d'autres représentants.

Le logiciel

La CM est programmée dans des langages classiques comme Fortran, muni d'extensions parallèles, notamment de tableaux distribués. *Lisp et CM-Lisp ou encore C* sont des extensions de langages offrant des primitives de traitement parallèle des données. Les compilateurs produisent des programmes en assembleur CM, qui permet de profiter de façon transparente du parallélisme et masque l'implantation physique de la CM. La CM peut supporter une à quatre machines hôtes qui s'en partagent l'accès.

Ce modèle n'est cependant pas répandu auprès des programmeurs, et ce sont toujours des outils automatiques qui déterminent la distribution des données des programmes écrits dans des languages classiques.

3.1.2 Parallélisme des traitements

3.1.2.1 Principe

L'ensemble des traitements est découpé, à une granularité plus ou moins fine correspondant au grain de parallélisme, en unités. Une relation de dépendance entre ces traitements est induite de la solution : elle indique les couples de traitements qui possèdent une dépendance de données. Cette relation est un ordre partiel sur les traitements et mène à plusieurs exécutions parallèles possibles.

La décomposition en unités de traitement permet de mettre en évidence des propriétés structurelles du problème ou d'utiliser les caractéristiques de la machine. Cet aspect a été étudié en 2.1.

La communication entre unités de traitement amène à distinguer deux sous-classes de machines MIMD : celles à mémoire partagée et celles communiquant par échanges de messages. Nous nous intéressons plus particulièrement à la deuxième, que nous allons illustrer par la machine V256 du projet Victor d'IBM [Shea89]. Dans cette sous-classe, les unités de traitement sont exécutées par les nœuds de la machine, et les données sont partagées statiquement et échangées. Ces échanges sont supportés par le réseau d'interconnexion entre nœuds. Cette approche est la plus répandue pour les machines parallèles.

3.1.2.2 Le matériel

La machine V256 est formée d'une grille 16 × 16 de processeurs, soit un total de 256 processeurs. Chaque processeur exécute son propre flot de contrôle d'instructions sur des données propres non partagées. Les instructions et les données sont contenues dans la mémoire locale de chaque processeur. Les processeurs sont des transputers T800, et communiquent chacun avec quatre voisins (voir figure 3.2).

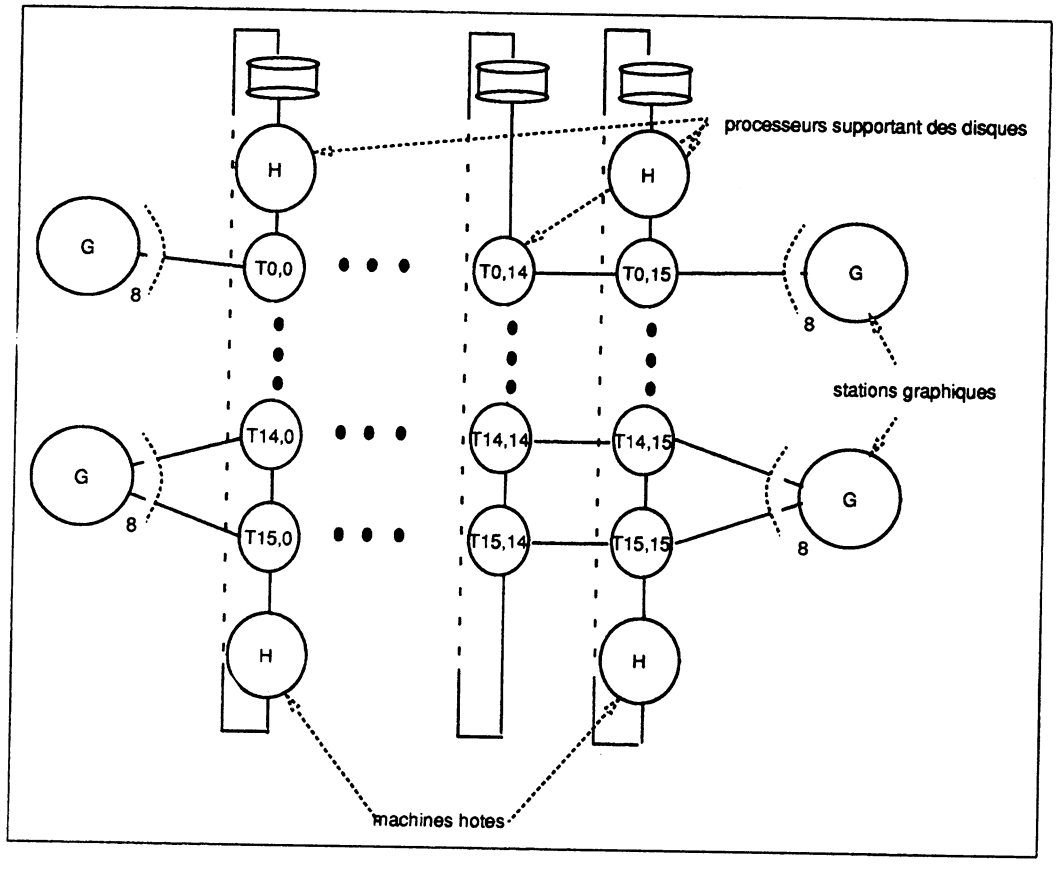


Figure 3.2: Architecture du V256

Choix de la topologie

Dans le choix de la topologie du V256, le faible nombre de liens du processeur utilisé est une restriction. Cependant, ses constructeurs ont préféré une topologie régulière plutôt qu'une topologie irrégulière de plus faible diamètre.

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 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 demande 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. Du fait que le bus soit global, nous pouvons prévoir que l'extension de la machine sera limitée par sa bande passante.

Choix des unités périphériques

Sur la base d'expériences réalisées sur d'autres machines, où les entrées-sorties disque constituaient un goulot d'étranglement, les concepteurs du V256 le munissent d'un disque par élément vertical, ainsi que l'illustre la figure 3.2. Du même coup, ils assurent la possibilité à tout utilisateur, au prix de quelques restrictions quant à l'allocation des sous-réseaux de processeurs, de disposer d'un accès à un serveur disque. Une contrainte est de ne pas traverser d'autres sous-réseaux. Le débit cumulé des disques est de 16 Mo/s, au lieu d'un débit cumulé idéal théorique de 100 Mo/s.

Utilisation du parallélisme

La machine est partitionnée entre différents utilisateurs qui peuvent mettre en œuvre des environnements de programmation différents. Aucun modèle 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 gestion des 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.

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 est consommée pour l'acheminement de messages. D'autre part, un réseau d'interconnexion dédié au support système d'observation, de déboggage 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 [Waille90] que nous étudierons en 4.1.2, proposent un tel réseau, mais découpé selon le partitionnement de la machine.

Enfin, la machine est partitionnée entre des utilisateurs mettant en œuvre des environnements de programmation différents.

3.2 La communication dans les machines parallèles

La communication, dans les machines parallèles comme dans les machines mono-processeur, peut être mise en œuvre par l'intermédiaire de deux techniques : la mémoire partagée et l'échange de messages. Ces deux techniques sont supportées par un réseau d'interconnexion entre processeurs et mémoire dans le premier cas, entre processeurs dans le second.

3.2.1 La communication par mémoire partagée

Les réseaux d'interconnexion des machines communiquant par mémoire partagée sont des graphes permettant la communication entre les ensembles de processeurs et de mémoires.

Cette sous-section sera illustrée par un exemple : le projet Research Parallel Processor Prototype (RP3) [Pfister86].

3.2.1.1 Le Research Parallel Processor Prototype (RP3)

Le projet RP3 est la matérialisation des hésitations d'IBM quant à la tendance future des machines parallèles. Que les machines parallèles soient amenées à se développer, nul doute. Mais sur quel type d'architecture faut-il investir ? Comment seront programmées ces machines ? De ces questions, parmi d'autres que se pose le monde scientifique dans sa majorité, naît le projet RP3.

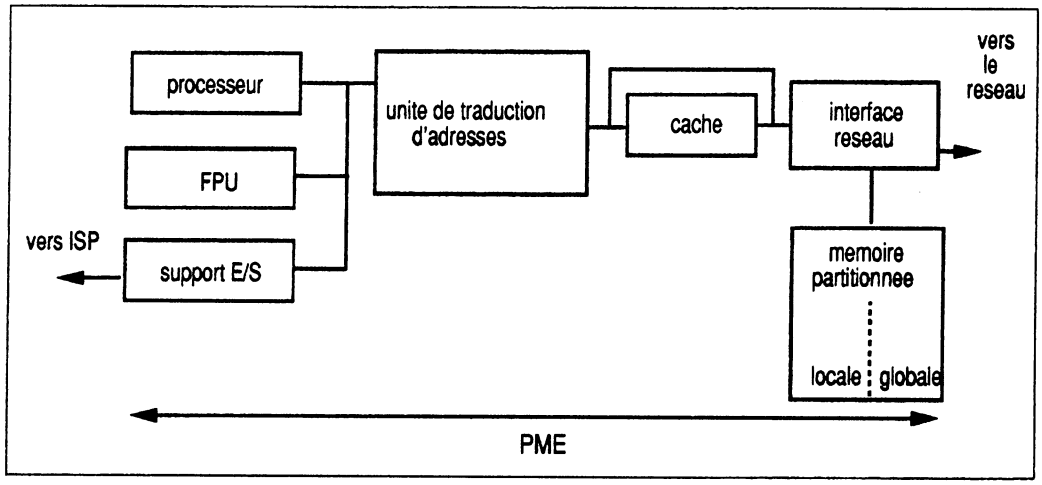


Figure 3.3: Un nœud de la machine RP3

Il s'agit d'étudier et de réaliser, en collaboration avec le projet Ultracomputer [ELS88a,ELS88b] du Courant Institute de New York University, un prototype de machine alliant la mémoire partagée à l'échange de messages, et ceci de façon paramétrable afin de faciliter l'expérimentation. Ainsi, le RP3 couvre le spectre allant des machines à mémoire partagée à celles à mémoire strictement privée. Un effort important, où se révèlent beaucoup de points d'interrogation, est consacré au support d'initialisation, d'analyse, de contrôle et d'entrées-sorties. Nous nous intéressons ici au réseau d'interconnexion.

La machine (voir figure 3.3) est constituée d'un ensemble de modules reliés par un réseau de commutateurs de connexion. Chaque module comporte un ensemble de nœuds (*"Processor Memory Element"* ou PME) de calcul, tous reliés à un processeur d'entrées-sorties (*"I/O and Support Processor"* ou ISP). Les ISP permettent différentes connexions vers divers périphériques (disques, terminaux) et machines hôtes (IBM S/370).

3.2.1.2 Structure d'un nœud

Chaque nœud, ou PME, comporte un processeur général, une mémoire partitionnée en deux sous-ensembles, l'un de mémoire globale, l'autre de mémoire locale, et une mémoire cache. La frontière entre les deux parties de la mémoire est modifiable.

Les adresses émises par le processeur vers la mémoire sont interprétées par l'interface réseau qui peut les aiguiller soit vers la portion de mémoire locale, soit vers le réseau de commutation entre modules. Après passage dans le réseau, l'adresse sera transmise à l'unité d'interface réseau d'un PME où elle identifiera une case de sa portion de mémoire partagée. L'unité d'interface des entrées-sorties permet d'adresser un ISP partagé entre 8 PME, qui forment un module.

Le nombre de processeurs pouvant accéder simultanément à la même adresse mémoire est important (il peut y en avoir 512). De ce fait, il peut y avoir congestion lors des accès à des adresses mémoire identiques. Des techniques de prévention de congestion sont mises en œuvre. La mémoire globale est largement entrelacée, des pages consécutives se trouvant sur des PME différents. De plus, les accès consécutifs aux adresses d'une page sont réarrangés par une fonction de dispersion ("hashing").

Le réseau d'interconnexion

Le réseau d'interconnexion est divisé en deux composantes qui accomplissent des fonctions différentes.

Les réseaux Banyans autorisent un nombre de processeurs différent de celui des mémoires. Chaque processeur est la racine d'un arbre de connexions dont toutes les mémoires sont les feuilles. Les arbres de connexions de processeurs différents ne sont pas indépendants ; le réseau est donc bloquant. Il existe donc un chemin unique pour tout couple processeur-mémoire.

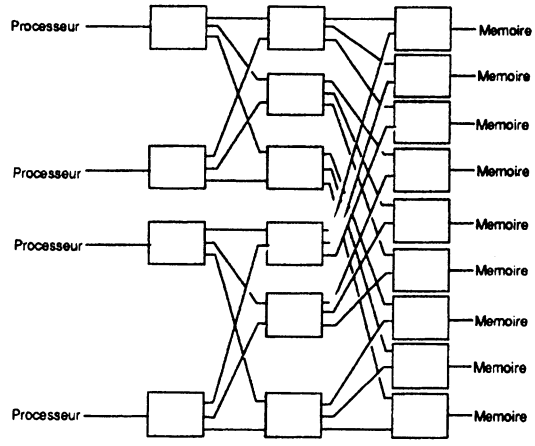


Figure 3.4: Réseau Banyan, appliqué à la connexion processeurs-mémoires

La première composante est un réseau SW Banyan (voir figure 3.4) rectangulaire. Il réalise la connexion d'un PME vers une mémoire avec un temps de latence faible.

La seconde composante est un réseau Omega (voir figure 3.5) qui permet la synchronisation des accès concurrents entre processeurs. Il combine des accès concurrents à la même adresse mémoire et permet des accès et modifications atomiques ("fetch-and-operate").

Le choix entre les deux réseaux est déterminé par le type d'accès. Lorsqu'une adresse est soumise à l'un des deux réseaux, elle traverse des commutateurs en pipeline vers le PME destination. Lorsqu'un étage du réseau bloque, la requête est conservée dans les différents

Le réseau Omega offre un seul chemin possible entre tout couple processeur-mémoire. Il est bloquant, tous les chemins ne pouvant être réalisés simultanément, et souffre d'une forte probabilité de blocage entre les chemins. Une solution au problème de blocage est dans la *combinaison* des accès à la même adresse mémoire.

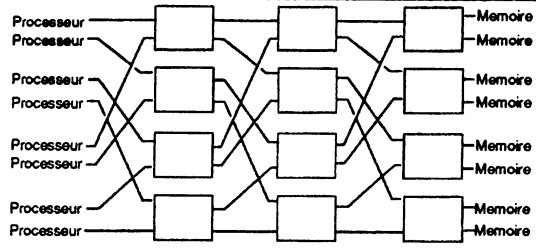


Figure 3.5: Réseau Omega appliqué à la connexion processeurs-mémoires

étages déjà traversés en attendant la libération de la ressource suivante occupée. L'ensemble de la requête passe par le même chemin que la tête, réalisant ainsi un mélange de commutation de circuit et de découpage en paquets.

Le réseau à faible délai de latence peut soutenir un débit de 50 Mo/s par connexion de processeur à mémoire, et fournir 128 connexions au total, chacune fonctionnant en requête et réponse, soit un débit total maximal de 12.8 Go/s. Le délai de latence est de 500 ns pour une requête de 8 octets.

Le réseau combinant les accès est une succession de commutateurs, chaque étage gérant une file d'attente des références à des adresses identiques, qu'il peut combiner. Le débit total est identique à celui du réseau à faible délai, mais le temps d'attente est plus important.

3.2.1.3 Environnement logiciel

Sans certitude sur les langages de programmation et les supports systèmes pour leur exécution sur une machine parallèle, RP3 se contente de langages et de systèmes d'exploitation classiques. Le système d'exploitation initial opérant sur les nœuds est Unix BSD 4.2 dans une version parallélisée et régulant la charge de la machine. Au niveau utilisateur, le parallélisme apparaît par l'usage de la mémoire partagée entre processus et une primitive permettant de créer plusieurs processus en parallèle.

Un objectif primordial est de pouvoir récupérer les programmes existants écrits en langages C, Fortran ou Pascal. Il s'agit de leur faire bénéficier du parallélisme sans les modifier. D'autre part, les langages sont modifiés de façon à offrir des extensions pour le parallélisme. Celles-ci consistent à distribuer des données aux PME, et les répartir en données locales et globales, et aussi à traiter les boucles en parallèle.

3.2.2 La communication par échange de messages

Ce type de communication met en jeu des paires de processeurs, et les réseaux correspondants sont quasiment toujours symétriques. Le choix de la topologie du réseau d'interconnexion est fondamental pour les performances et la facilité d'utilisation de la machine. De cette topologie dépendent la distance entre processeurs et le taux d'utilisation des connexions.

Deux objectifs prévalent dans le choix d'une topologie : minimiser la distance moyenne entre processeurs et uniformiser le taux d'utilisation des connexions. Ceci peut être obtenu par différentes approches.

3.2.2.1 Les réseaux maillés

L'arbre, la grille et l'hypercube sont des topologies classiques. Elles peuvent être combinées dans des assemblages plus complexes.

L'hypercube

Il réalise un diamètre de l'ordre du logarithme du nombre total de processeurs et une topologie symétrique. Les techniques de routage sont largement connues et optimisées pour cette topologie. C'est une structure hiérarchique dont l'extension se fait en doublant le nombre de processeurs, ce qui constitue un incrément considérable, en général trop important pour la puissance additionnelle souhaitée et le coût financier. D'autre part, l'ajout de processeurs augmente le degré des processeurs formant l'hypercube, ce qui n'est pas aisément réalisable physiquement.

L'adressage dans un réseau en hypercube utilise généralement les codes de Gray [SaSchu88], et permet de distinguer des dimensions et de regrouper les nœuds correspondant à une même dimension. On peut ainsi réduire la complexité du contrôle de la totalité de la machine en considérant des parties de d^n nœuds. Cependant, cette topologie ne distingue aucun processeur pouvant servir aux E/S disque ou au support pour usagers interactifs.

Si des programmes application se satisfont d'une topologie en hypercube, le support du système d'exploitation peut nécessiter une autre topologie, notamment en fonction des unités périphériques (disques, terminaux) disponibles. Le support système (analyse, services, entrées-sorties) ne peut se distinguer de l'exécution normale des applications et doit s'exécuter sur les nœuds ordinaires. On observe d'ailleurs que dans la quasi totalité des machines de cette topologie, le support système est fourni par une machine hôte et que l'exploitation du cube se limite au chargement de programmes et à une bibliothèque de communication.

La grille

La grille de dimension n, parfois refermée en tore, présente un diamètre de l'ordre de la racine énième du nombre total de processeurs. C'est une topologie symétrique pour laquelle de nombreux algorithmes de routage existent. Son extension peut se faire par ajout d'une ligne ou d'une colonne, c'est à dire un nombre de l'ordre de la racine énième du nombre total de processeurs. Le degré des processeurs reste constant en cas d'extension. De plus, si elle n'est pas fermée en tore, la longueur des supports de communication entre composantes est constante.

Il est difficile de distinguer des sous-réseaux de processeurs, mis à part le regroupement par lignes ou colonnes qui est très peu flexible et rompt la topologie globale. Les remarques quant au support système énoncées dans le paragraphe précédent à propos de l'hypercube sont valables pour la grille.

La maintenance de politiques générales efficaces passe par une connaissance précise de l'état global de la machine, qui n'est observable qu'au prix de communications et de synchronisations multiples. Dans les réseaux à grille supportant des systèmes d'exploitation, par exemple le V256 du projet Victor décrit en 3.2, chaque processeur est contrôlable directement par un dispositif indépendant de la grille, et peut accéder à des périphériques par d'autres supports de communication.

Les systèmes d'architecture iWarp [Bork89] illustrent les réseaux plans à dimension 1 ou 2.

L'architecture iWarp

La machine iWarp 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 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 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.

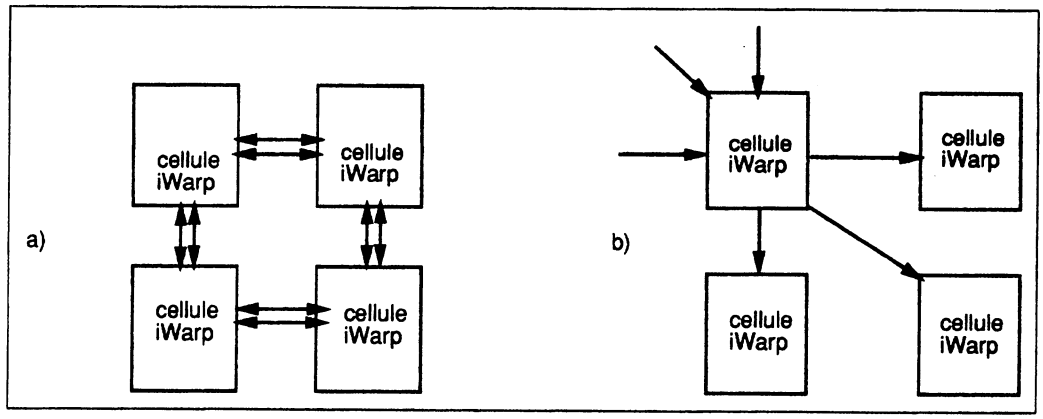


Figure 3.6: Différents réseaux pour la machine iWarp

Le matériel fournit le support pour le routage *"wormhole"* [DaSe87] 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'apparentent 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 du message est lu et immédiatement réémis sur le bus logique suivant sur son chemin (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. 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 (voir figure 3.6 a). Dans les seconds, les ports sont découplés et l'on peut ainsi réaliser un treillis hexagonal par exemple (voir figure 3.6 b).

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 est 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.

Une machine iWarp peut servir d'accélérateur de calcul associé à une station hôte, mais peut aussi être autonome. A cet effet, un accès direct à la mémoire locale de chaque nœud est disponible, permettant d'interfacer des bus standards ou des unités périphériques (disques, terminaux etc). Les entrées-sorties sont donc fortement couplées à chaque nœud, permettant ainsi des usagers interactifs.

Les topologies mixtes

Beaucoup de machines (voir les projets Eden[Lazow81], CM* [OSS80] et Suprenum [WSP90]) sont construites suivant une topologie mixte, regroupant les nœuds en "clusters" soumis au contrôle d'une unité assurant en même temps un support pour l'analyse et l'instrumentation. D'autre part, des voies d'entrées-sorties à très haut débit sont disponibles (voir iWarp dans les paragraphes précédents).

Le réseau idéal est le crossbar, qui permet à chaque processeur d'accéder à toute la mémoire. Son inconvénient est son coût de réalisation élevé, et croissant fortement avec le nombre d'entrées ($O(N^2)$).

Les réseaux de Benès offrent plusieurs chemins entre tout couple processeur, mémoire. Ils sont réarrangeables, c'est-à-dire que toute nouvelle connexion est possible, mais éventuellement au prix d'un réarrangement des connexions déjà établies. Le temps d'établissement d'une configuration est plus important que dans le cas des réseaux Omega.

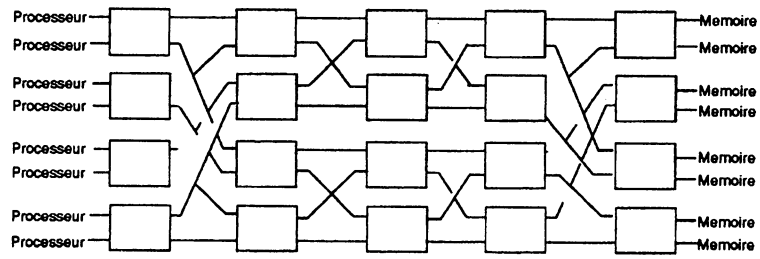


Figure 3.7: Réseau de Benès appliqué à la connexion de processeurs-mémoires

Le regroupement de processeurs en clusters vise à réduire l'ordre de grandeur des problèmes de communication. La topologie du cluster est parfois un bus, mais il se dessine une forte tendance au développement de réseaux de communication permettant une communication totale, réalisant une approximation des commutateurs crossbar (voir figure 3.7).

Fujimoto et Reed [ReFu87, p52–78] étudient les réseaux d'interconnexion à un étage d'un point de vue théorique. Quinze réseaux sont comparés selon plusieurs critères : le nombre de nœuds par rapport au nombre de connexions et de liens, l'incrément et le facteur d'échelle, la connectivité en arcs et en sommets, le diamètre, la distance moyenne entre sommets, et enfin le nombre moyen de visites d'un message à

un sommet. De cette étude, il ressort qu'en dehors de toute contrainte, l'hypercube à bus recouvrants ("spanning bus hypercube") est préférable pour tous les critères excepté le nombre moyen de visites. Le n-cube k-aire et le tore offrent de loin le meilleur débit mais la connectivité du réseau est limitée. L'hypercube binaire est intéressant lorsque la connectivité peut indéfiniment augmenter. Enfin, les réseaux asymétriques ne sont en général pas indiquées excepté pour des applications spécifiques.

Le nombre de connexions est en général limité à une dizaine, ce qui porte à adopter des topologies pour lesquelles le degré de chaque sommet n'est pas une fonction directe du nombre total de processeurs pour des machines de moyenne et grande tailles ($\geq$ 100 et $\geq$ 1000 processeurs).

Le projet Supernode propose un composant de reconfiguration dynamique, qui permet d'adapter la topologie physique au problème traité. En effet, il n'existe pas de topologie idéale pour l'ensemble des applications. L'ajout de nœuds se fait alors par unités dans un cluster, ou par grappes regroupées en clusters. La distance entre processeurs peut toujours être rendue égale à l'unité.

La famille de machines à base de processeurs T9000 [Pount90] adopte une autre approche, en intégrant des processeurs de communication et de routage. Les performances en communication sont alors telles que l'on espère que leur variation en fonction du diamètre du réseau seront négligeables (on passe de 28 Mo/s à 63 Mo/s pour un diamètre passant de 6 à 14, soit des rapports de $\frac{28}{63}$ et $\frac{27}{63}$).

3.3 Les nœuds des machines parallèles

Les nœuds des différentes machines parallèles sont bâtis suivant une architecture classique. Soucieux de bénéficier des programmes et outils de développement déjà disponibles pour des processeurs conventionnels, la large majorité des machines parallèles propose un nœud bâti autour d'un processeur traditionnel. Cependant, quelques machines utilisent des processeurs dédiés réalisant en matériel les fonctionnalités les plus critiques pour l'efficacité globale.

3.3.1 Fonctions générales

Les communications par mémoire partagée et par échanges de messages sont toutes deux utilisées. L'une et l'autre peuvent être mises en œuvre sur des architectures à mémoire partagée ou privée (directement intégrée sur la puce). Plusieurs machines les utilisent simultanément, en distingant une mémoire privée pour chaque nœud

et une mémoire globale commune. Chaque nœud dispose d'une mémoire locale, qui peut être privée ou partagée. Les temps d'accès des mémoires de taille importante adressées simultanément par plus d'une dizaine de processeurs étant très élevés par rapport au temps d'accès d'une mémoire privée, les processeurs disposent souvent d'une mémoire privée très rapide. L'accès à une mémoire partagée se fait en réduisant la probabilité d'accès simultané par des techniques d'entrelaçage, réarrangement ou combinaison entre autres.

Les machines sans mémoire commune disposent d'un processeur de communication. Il accomplit des fonctions plus ou moins sophistiquées, allant du routage à la simple copie de l'information en mémoire. Il réalise un protocole mettant en jeu des entités actives du processeur central. Ce dernier est ainsi déchargé des opérations de désignation, d'initialisation, de routage et de terminaison des communications.

Un processeur de calcul flottant peut compléter chaque nœud. La puissance de chaque nœud est celle des processeurs classiques, de l'ordre de 10 à 20 Mips pour 2 à 20 Mflops. Les processeurs de communication sont moins conventionnels et débitent entre 3 Mo/s et 320 Mo/s, pour un délai de latence extrêmement variable, de l'ordre de 100 ns à quelques microsecondes.

3.3.2 Débit d'entrée-sortie et puissance de calcul

Par rapport aux systèmes répartis, les machines parallèles sont en général fortement couplées de par leurs réseaux d'interconnexion. Selon l'application en cours d'exécution, la communication entre processeurs ou entre processeurs et mémoires peut demander l'échange très fréquent de messages courts (quelques dizaines d'octets), ce qui se traduit par la nécessité de délais de latence faibles, de larges bandes passantes et de possibilités de multiplexage. Les entrées-sorties au contraire nécessitent des échanges de messages moins fréquents de messages dont le grain, pour les accès aux fichiers, est généralement un multiple de la taille du bloc disque (en moyenne 8 Ko [RST89]).

L'utilisation du même réseau pour les deux services pénalise l'un et l'autre. Cependant, la co-existence de deux réseaux distincts est coûteuse, et beaucoup de machines les confondent. Les accès aux périphériques sont alors acheminés par des processeurs intermédiaires, réduisant ainsi le débit effectif des entrées-sorties. Pour tenir compte de ce phénomène, il faut modifier la politique des entrées-sorties [MagMun89].

D'un autre côté, les périphériques dits "caractères" ne se comportent pas de la façon décrite plus haut : l'unité d'information est de l'ordre de la chaîne de caractères (moins de 80 caractères). La conception du réseau d'entrées-sorties en est compliquée d'autant.

Projet	Mips/proc	E/S	Mips total	E/S disque
Linda	68020 >>	0.5 Mo/s	×	×
iWarp	20 Mips <<	320 Mo/s	1.2 Gips	bus standard
Connection Machine	non significatif	×	2.5 Mips << 20 Gflops	210 Mo/s
Medusa	LSI11	1.15 Mo/s 0.43 Mo/s 0.17 Mo/s	×	×
RP3	PC RT <<	13 Go/s	1.3 Gips >> 800 Mflops	192 Mo/s
Supernode	10/15 Mips	10 Mo/s	$10 \times N$ Mips	×

comparaison par rapport à la puissance théorique de crête
en traitement et communication
>> signifie très supérieur à,
× représente des valeurs non communiquées

Table 3.1: Puissances comparées

Les processeurs de calcul

Les processeurs de calcul, pour débiter les puissances dont ils sont théoriquement capables, doivent disposer d'un flux de données correspondant à leur vitesse de traitement. Pour cela, il faut maintenir un flux d'entrées-sorties et de communications assurant la disponibilité des données et des programmes nécessaires à la suite du traitement.

Le débit d'entrées-sorties requis varie fortement avec les applications et les modèles de programmation, mais un débit de crête théorique peut être estimé à partir de la puissance de crête théorique du processeur de calcul. Par défaut d'informations supplémentaires sur le comportement des programmes parallèles, nous adoptons la règle traditionnelle 1 Mo/s et 1 Mo de mémoire pour chaque Mip comme critère d'équilibre entre entrées-sorties et puissance de calcul. Il faut cependant se rappeler que cette règle a été constatée pour des machines mono-processeur conventionnelles. D'autre part, nous ne nous intéressons pas à la mémoire dans ce rapport.

En considérant le rapport de la puissance par processeur au débit par processeur de communication résumé en table 3.1, il apparaît que le débit est très largement supérieur à la puissance de calcul. Ce rapport n'est cependant pas déterminant, car les processeurs de communication ont aussi, le plus souvent, la charge d'acheminer des messages pour le compte d'autres processeurs.

Pour affiner notre mesure, considérons que, pour émettre un message, chaque processeur utilise les services d'un certain nombre de processeurs de communication du réseau. Ce nombre est en moyenne égal à la distance moyenne entre nœuds. Etudions le cas de iWarp dans une configuration de tore. D'après [ReFu87], la distance moyenne

pour le tore est donné par la formule suivante, pour une distribution uniforme de messages :

$$\overline{d} = \begin{cases} \frac{2\times N\sqrt{N}}{4(N-1)} & \sqrt{N} \text{ pair, } \simeq \sqrt{N}/2 \text{ pour } N \text{ élevé} \\ \frac{2\times(N-1)\sqrt{N}}{4(N-1)} & \sqrt{N} \text{ impair, } = \sqrt{N}/2 \end{cases}$$

Le débit de chaque nœud est donc à diviser par cette distance moyenne pour trouver la puissance de communication effectivement utilisée par le nœud. Cherchons le nombre de nœuds pour lequel le rapport s'inverse :

$$20 = \frac{320 \times 2}{\sqrt{N}} \Rightarrow N = (\frac{320 \times 2}{20})^2 = (32)^2 = 1024 \text{ processeurs}$$

Ce nombre est justement celui annoncé pour la plus grosse machine iWarp. iWarp présente l'un des processeurs de communication les plus puissants, et l'on peut donc en conclure que les autres topologies en grille ou tore à plus d'un millier de processeurs présentent un déficit en puissance de communication, pour une répartition des communications homogène. Une fois de plus, la solution consistant à diviser la machine en unités plus facilement contrôlables se révèle appropriée.

Les entrés-sorties

Etudions à présent les possibilités en entrées-sorties. Ce sujet est souvent délaissé par les constructeurs de machines parallèles, qui se contentent de solutions classiques. Remarquons cependant que, après une première version relégant les entrées-sorties aux machines hôtes, la Connection Machine consacre un effort important aux entrées-sorties disques, avec un débit élevé. Plusieurs unités de stockage fonctionnent en parallèle et peuvent débiter jusqu'à 210 Mo par seconde.

Il faut à nouveau tenir compte du fait que chaque nœud n'a pas toujours directement accès aux supports d'entrées-sorties. Il peut être nécessaire d'acheminer l'information à travers un réseau d'interconnexion avec des supports d'entrées-sorties. Ce réseau se réduit souvent à un bus partagé par les nœuds d'un cluster. Pour les machines iWarp (resp. RP3), on a une puissance de calcul de crête totale de 1.2 Gips (resp 1.3 Gips), pour des débits de l'ordre d'une connexion par accès direct à la mémoire à travers une interface de bus conventionnelle (resp 192 Mo/s). La règle de 1 Mips pour 1 Mo/s est loin d'être respectée.

Elle doit être atténuée en fonction de la puissance de communication déjà disponible. La puissance de calcul est supérieure d'un ordre de grandeur au débit en entrées-sorties. De nombreux travaux sont

menés pour structurer les systèmes de fichiers de façon à prendre en compte ce rapport de puissances [RST89].

Pour ce qui est des entrées-sorties par caractères, les applications interactives requièrent un couplage fort entre le périphérique et le nœud. Si l'information doit transiter par d'autres nœuds supportant d'autres applications, le temps de réponse devient trop important. Deux solutions sont généralement adoptées.

La première consiste à établir un couplage fort entre terminaux et nœuds. Dans ce cas, le réseau perd sa symétrie car quelques nœuds ont un rôle différent des autres. Cette approche est adoptée dans iWarp. La seconde consiste à disposer de stations de travail supportant les travaux interactifs. Les traitements plus importants sont transmis à la machine parallèle. Cette solution est adoptée par RP3.

3.3.3 La résistance aux pannes

La résistance aux pannes, qui semble devoir aller de soi dès lors que l'on dispose de ressources multiples, est rarement abordée dans ces machines. Une première justification, la fiabilité supposée du matériel, en a été fournie dans le premier chapitre. La résistance aux pannes dans les machines parallèles est un problème très complexe. En effet, la multiplication des processeurs et des dispositifs de communication augmente d'autant les sources possibles de pannes.

La solution en général adoptée consiste à offrir une primitive permettant d'enregistrer l'état d'une application, à analyser dynamiquement la machine pour détecter des mauvais fonctionnements éventuels, et reprendre l'exécution à partir du dernier état stable enregistré. La duplication des ressources allouées à chaque application est très peu utilisée dans les machines et systèmes parallèles à usage général. En effet, la consommation de ressources est alors prohibitive.

Cette solution requiert la disponibilité d'un support d'information fiable, de capacité importante, et la possibilité de l'accéder dans des temps suffisamment courts n'affectant pas notablement la performance globale de la machine. Ces supports sont en général des disques accédés par un réseau à très haut débit. Ce réseau est le plus souvent différent du réseau d'interconnexion entre nœuds car il est caractérisé par de gros volumes d'informations ($> 1\ Ko$) mais peu fréquemment échangés par rapport à la fréquence des interactions entre nœuds dans le réseau.

3.4 Conclusion

En adoptant une approche essentiellement pragmatique, illustrée par l'expérience d'un éventail représentatif des machines parallèles const-

ruites ou à l'état de prototypes, ce chapitre a posé les fondements pré-requis pour l'architecture et le support système de très bas niveau dans les machines parallèles.

Nous résumons les enseignements de cette étude en cinq points :

1. Chaque machine présente un certain nombre de caractéristiques correspondant à une classe d'applications donnée. Il est imprudent d'extrapoler des performances d'une machine en multipliant son nombre de processeurs : suivant la topologie de la brique de base, les performances ne croissent pas proportionnellement au nombre de processeurs.

2. Le support d'exécution, qui ne s'étend en général pas très bien à un très grand nombre de processeurs (de l'ordre du millier), devient un goulot d'étranglement insupportable. Pour conserver des complexités contrôlables, les machines sont souvent subdivisées en sous-ensembles ou clusters, permettant de mettre en œuvre des politiques selon des niveaux d'abstraction de plus en plus élevés. Chaque politique peut correspondre à un sous-système. Il n'existe pas (encore !) de modèle de programmation unifiant tous les programmes parallèles de façon satisfaisante.

3. Une machine parallèle supporte non seulement des applications dont les besoins induisent une topologie particulière, mais aussi du logiciel opératoire et de contrôle qui demande une topologie en général centralisée. La nécessité de cette topologie, requérant souvent des entrées-sorties, est illustrée dans le traitement de la tolérance aux pannes. On distingue souvent les processerus supportant des programmes systèmes (le réseau système) de ceux supportant des programmes utilisateurs (réseau utilisateur).

4. Un aspect important d'une machine parallèle est sa capacité à s'adapter à la topologie requise par une application donnée. Ceci est en général obtenu en fournissant des processeurs et un réseau d'interconnexion virtuels.

5. Le dimensionnement des puissances de traitement, de communication et d'entrées-sorties périphériques est fondamental. Il doit être effectué de façon à éviter qu'un processeur devienne un goulot d'étranglement.

Les tableaux 3.2 et 3.3 présentent les caractéristiques de quelques machines. Nous n'avons pas présenté CM* [OSS80], Eden [Lazow81] et Linda [ACG88], qui faisaient partie des machines étudiées mais qui, pour des raisons de place, n'ont pas été détaillées ici. Une machine non spécialisée doit offrir les mécanismes nécessaires au support de différents modèles, l'aspect politique ne devant intervenir qu'en fonction de l'application parallèle que l'on souhaite exécuter.

×	Eden	Linda	iWarp	C-Machine
le nœud de la machine :				
proces-seur	-général -communication -d'E/S -disques -ttys	-général -communication	-général -communication	élémentaire
mémoire	locale partagée	locale	locale	locale
réseau	bus multiples	bus multiples	registres	grille
expansion	1 à 4			
réseau d'interconnexion :				
topologie	-diffusion -réseau local -statique	-diffusion -grille -statique	-réseau maillé -configurable	-hypercube -n-grille
$\overline{d}$	1	1	$\frac{N \times \sqrt{N}}{2(N-1)}$ (tore)	non significatif
C-arcs	1	$\sqrt{N}$	4	non significatif
C-somm	N	N	4	non significatif
Max-som	$4 \times L$	$2\sqrt{N}+1$	$2\sqrt{N}+1$	$(\log_n N+1)^n - N$
disques, tty				oui
logiciel :				
noyau	oui	classique	machine hôte	machine hôte
serveurs	oui	classique	machine hôte	machine hôte
entités	objets	processus	"data flow" ou tâches	"data parallelism"
opérations	invocation	in, out, rd, eval		

C-arcs : connectivité en arcs
C-somm : connectivité en sommets
$\overline{d}$: distance moyenne entre sommets
L : nombre maximal de stations sur réseau local à diffusion
Max-som : nombre maximal de sommets
N : nombre de nœuds de la machine

Table 3.2: Récapitulatif de caractéristiques (à suivre)

×	CM*-Medusa	RP3	Supernode
le nœud de la machine :			
processeur	général	général	-général -communication -support multi-tâches
mémoire	locale partagée	locale partagée	locale
réseau	bus	bus	interface mémoire
expansion	10 (L)		
réseau d'interconnexion :			
topologie	-diffusion -bus hiérarchique -statique	-mémoire partagée -réseau double configurable	-réseau maillé -commutateurs hiérarchiques -reconfigurable
$\overline{d}$	1	1	1 et variable
C-arcs	nombre clusters		variable
C-somm	$L\times$ nb clusters	technologie	N
Max-som	1, cluster	1 ou 8	1, cluster
disques			
tty		hôte	oui
logiciel :			
noyau	oui	classique	oui
serveurs	oui	classique	oui
entités	"tasks"	classique	3 niveaux de processus
opérations	invocation	boucles parallèles	échanges de messages

Table 3.3: Récapitulatif de caractéristiques (suite et fin)

Chapitre 4

La machine Supernode et les futurs transputers

Les Supernodes sont un exemple important de machine parallèle, et servent de support d'expérimentation dans le projet Parx. Nous présentons leur principe dans ce chapitre. La première section présente brièvement les machines Supernode, dont l'une des caractéristiques est la possibilité de reconfigurer leur réseau d'interconnexion. La seconde section est une étude critique des transputers et des Supernodes, qui aboutit à la présentation d'une nouvelle famille de transputers.

4.1 Architectures reconfigurables

Le réseau d'interconnexion idéal permettrait à chaque processeur de disposer d'une connexion (physique) directe avec tout autre processeur. La topologie en bus réalise ce diamètre unitaire. Cependant, le nombre de processeurs, pour un niveau de performance acceptable, est très limité (quelques dizaines), à cause de problèmes de congestion lors de l'accès au bus. D'un autre côté, les composants matériels ont un nombre de pattes limité, et on ne peut les connecter tous simultanément pour plus de la dizaine de processeurs.

Les réseaux "crossbar" réalisent une correspondance bi-univoque entre leurs entrées et leurs sorties. La fonction de correspondance pouvant être modifiée en fonction des besoins de communication, on peut alors adapter le réseau physique à la topologie d'un problème. C'est l'approche adoptée dans Supernode.

Les processeurs sont tous reliés à un composant réalisant un réseau d'interconnexion dynamiquement reconfigurable à base de crossbars.

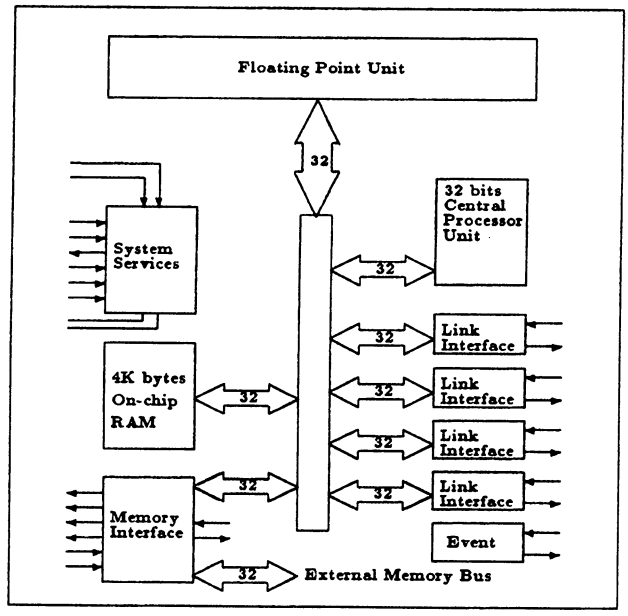


Figure 4.1: Schéma d'un transputer T800

4.1.1 Le transputer

Le transputer, illustré en figure 4.1, est un exemple de processeur à partir duquel on peut bâtir une architecture de machine Supernode. Les processeurs de la famille transputer possèdent chacun une unité de traitement, d'une puissance annoncée à 10 Mips pour le T800, une unité de communication, une unité de calcul flottant, une petite mémoire (2 Ko) sur le composant, et une interface pour une mémoire externe pouvant adresser 4 Go (voir figure 4.1).

L'unité de communication dispose de quatre liens série bi-directionnels, offrant chacun un débit de 20 Mbits/s en "full duplex". Chaque lien peut être directement relié à un autre lien transputer, ou bien à d'autres types de périphériques en utilisant un convertisseur. L'interface de communication place résolument le transputer dans la catégorie des processeurs pour réseaux point-à-point.

Le modèle de programmation occam, qui est sous-jacent dans chacun des choix de construction, impose l'absence de mémoire partagée entre processus, et l'échange de messages comme paradigme de communication. Le constructeur n'a cependant pas pris en compte l'exécution de programmes occam sur plusieurs processeurs. Le support offert à cet effet est très limité. Ceci est valable pour les communications et l'exécution de processus. Par exemple, des processus transputers père et fils partagent une même mémoire. Cela interdit l'usage de certaines constructions entre processus se trouvant sur des processeurs différents.

Chaque processeur offre un support matériel pour la simulation de processus parallèles. Une abstraction de processus est définie et supportée. Chaque processus pseudo-parallèle (certains auteurs les dénomment "concurrents"), possède un descripteur qui contient son état et des informations de chaînage pour l'ordonnancement et le gestionnaire de l'horloge physique de sa priorité, basse ou haute.

La politique d'ordonnancement est complètement assurée par le matériel en réaction à l'utilisation d'instructions interruptibles et d'une horloge permettant d'allouer des tranches de temps aux processus de basse priorité. La commutation de contexte ne demande que la sauvegarde de quelques registres, ce qui permet un temps de commutation inférieur à la microseconde. Un ré-ordonnancement intervient lors des instructions de communication et lors de certaines instructions de rupture de séquence. Les processus de haute priorité possèdent un quantum infini, et ne sont suspendus que lors des instructions de comunication.

4.1.2 Le modèle de machine Supernode

Nous ne faisons ici qu'une description très sommaire des machines Supernode. Pour une description détaillée, le lecteur pourra lire [Waille90].

L'idée fondamentale du modèle Supernode est de permettre, à partir d'une brique de base simple et contrôlable, la construction de machines parallèles dans un large éventail de tailles et de performances, le tout pour un coût modeste.

La brique de base, illustrée par la figure 4.2, est un ensemble de processeurs connectés suivant une topologie reconfigurable et disposant des supports système et matériel nécessaires à une machine complète.

La construction de machines plus complexes est hiérarchique, chaque brique devenant le nœud de base d'une nouvelle machine, disposant de support système et matériel. L'on peut ainsi bâtir des machines comportant des milliers de transputers sans pour autant perdre le support système au niveau de la brique.

L'architecture de base de chaque nœud s'articule en une partie contrôle et une partie opératoire. Cette dernière est formée d'un ensemble de processeurs dits de travail, connectés à travers un commutateur programmable. Ces processeurs constituent la puissance de travail disponible aux programmes utilisateurs. L'ensemble processeurs de travail-commutateur est piloté par un processeur de contrôle. Chaque brique de base peut comporter jusqu'à 32 processeurs de travail.

La partie contrôle est de loin la plus originale du Supernode. Elle est composée d'un processeur de contrôle commandant un commutateur de liens. Elle comporte aussi un ou deux processeurs dits de service,

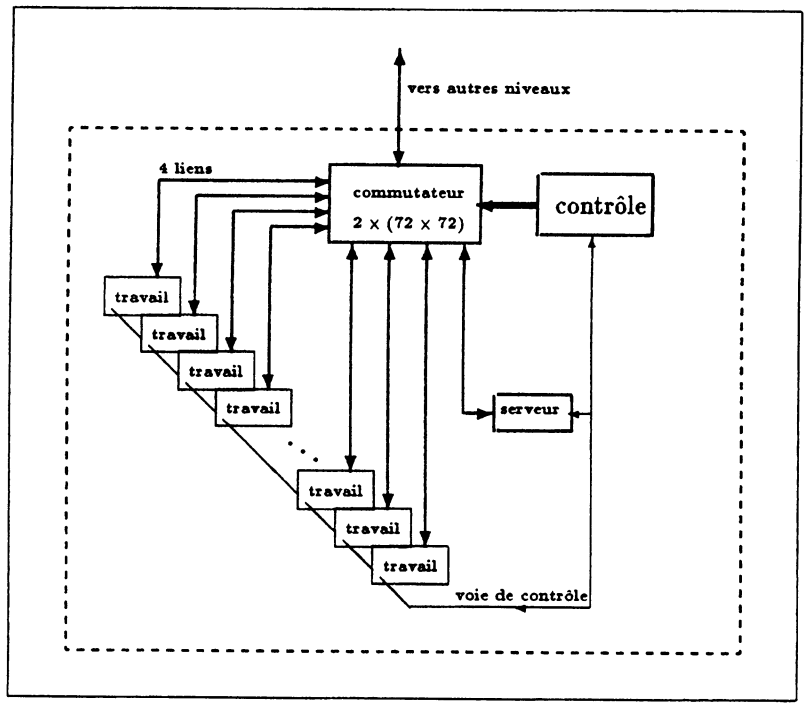


Figure 4.2: Un nœud du Supernode

couplés à des dispositifs périphériques (disques). Les liens des processeurs, tant de la partie opératoire que de la partie contrôle, sont reliés au commutateur.

Celui-ci permet de connecter dynamiquement n'importe quelle paire de liens, avec certaines contraintes de compatibilité, et par conséquent de reconfigurer le Supernode suivant la topologie la plus adaptée aux besoins d'une application. Toute topologie correspondant à un graphe étiquetté est accessible, à une renumérotation des liens près. La reconfiguration s'effectue en quelques microsecondes par simple programmation par le processeur de contrôle, et peut se dérouler sans perturber les processeurs dont les liens ne changent pas d'appariement. Ces caractéristiques permettent d'envisager des reconfigurations fréquentes de la machine sans perte d'efficacité.

Le processeur de contrôle est par ailleurs relié à chacun des processeurs de travail par une voie de contrôle selon un mode maître-esclave. Cette voie, dénommée bus de contrôle, offre des mécanismes de synchronisation et de signalisation entre processeurs maître et esclaves, et est essentiellement destinée à la circulation de messages liés à la reconfiguration : requête des esclaves, synchronisation entre processeurs impliqués dans une application, redémarrage d'un processus esclave etc.

L'architecture de la brique Supernode ressemble à celle d'un proces-

seur conventionnel destiné à fournir un support système. La partie contrôle peut être rapprochée du mode privilégié, la partie opératoire du mode esclave. Les entrées-sorties se déroulent en parallèle dans la partie contrôle. Le modèle de processus communiquant suivant un réseau complet se retrouve dans la partie opérative au travers de processeurs reliés par un réseau crossbar reconfigurable.

La machine Supernode peut donc être construite à partir d'un procédé récursif, le processeur abritant des processus communiquants, le Supernode à un étage des processeurs communiquants, les machines plus importantes des machines de taille inférieure etc. Paradoxalement, c'est au niveau du processeur —le transputer— que la récursion n'est plus respectée. Une nouvelle famille de transputers, baptisée T9000, vient pallier à ce manquement.

4.1.3 Les Supernodes à plusieurs niveaux

L'application du principe de récursivité crée des machines ayant de plus en plus de niveaux. Il est illustré par la figure 4.3. Chaque commutateur possède un certain nombre de liens connectables à un commutateur de niveau supérieur. Les commutateurs forment alors un réseau reconfigurable. Dans l'implantation actuelle, il s'agit d'un réseau de Clos qui est dynamiquement réarrangeable de façon bloquante. Les machines les plus importantes comptent aujourd'hui deux niveaux de commutateurs.

A cette complexité matérielle correspond une complexité logicielle décuplée. Le Supernode est un exemple de machine parallèle très puissante, mais sa programmation —et plus généralement la programmation parallèle— semble rebuter les programmeurs.

4.1.4 Environnement de programmation

Le Supernode, dans une configuration donnée, se comporte comme un simple réseau de transputers. Il bénéficie donc des environnements de programmation et de développement des réseaux de transputers.

Deux lignes se dégagent pour leur programmation. La première, et aussi la plus ancienne, est le "Transputer Development System" (TDS [Inmos88]) qui est un environnement construit essentiellement autour du langage occam. Il propose la chaîne "édition, compilation puis exécution", et, bien que très agréable pour le développement d'applications isolées, permet difficilement d'utiliser les résultats d'autres environnements.

La seconde ligne prend le contrepied de la première, en misant sur l'ouverture aux autres systèmes d'exploitation, éditeurs et compilateurs. Les langages de programmation sont plus divers et incluent

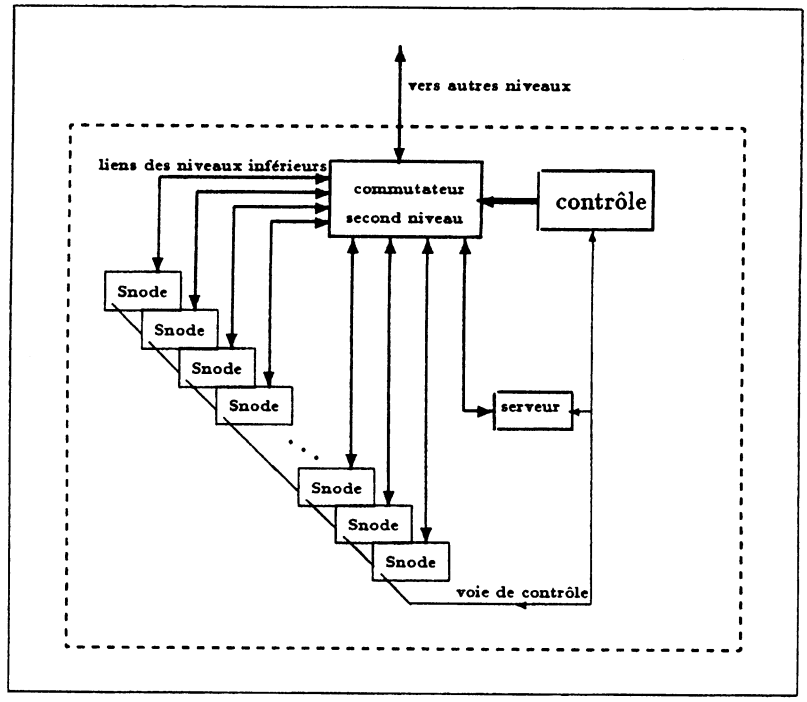


Figure 4.3: Un Supernode à plusieurs niveaux

occam, C, Fortran, Ada, SQL.

On peut compter Helios [Gar87], Idriss [King88] ou encore Trollius [Braner88] au rang des systèmes d'exploitation commercialisés.

Cependant, les deux approches n'utilisent pas toutes les capacités du Supernode, qui est très souvent considéré comme un banal réseau de transputers. La nouveauté des machines parallèles, le manque d'expérience des programmeurs, la complexité des problèmes posés par les algorithmes parallèles (les problèmes sont régulièrement np-complets) et la rusticité des outils proposés pour leur résolution, freinent les développements.

De véritables systèmes d'exploitation s'exécutant sur transputer et faisant usage des caractéristiques du Supernode sont à l'état de prototypes. Il faut mentionner *Mimics* développé par T. Botteri Corso au sein de l'équipe Flop [Botteri91], et qui est une adaptation, largement remodelée pour Supernode, du système d'enseignement Minix de Tanenbaum [Tanen87], et le noyau de système développé dans le projet Esprit Supernode II.

4.2 Nouvelles architectures transputer

4.2.1 Etude critique des systèmes à base de transputers

Ainsi que nous l'avons dit dans la présentation de ce processeur en section 4.1.1, il est à l'origine dédié à l'exécution de programmes écrits dans le langage occam. Victime de son succès, notamment auprès des numériciens, il est utilisé dans des applications de plus en plus générales, mettant en œuvre des concepts qui divergent de sa philosophie initiale. Il s'agit surtout du fort mouvement de standardisation autour de systèmes dérivés d'Unix, et de langages plus classiques (Ada, C, Fortran, Lisp, Prolog, SQL etc.).

Les critiques les plus fortes proviennent des constructeurs de systèmes d'exploitation et de supports de développement de programmes. Une large partie du peu d'empressement d'une certaine classe de programmeurs provenait en fait de l'environnement de développement initial, le "Transputer Development System" (TDS), bâti exclusivement autour du langage occam et n'offrant aucune ouverture aux applications et langages existants. Résumons les limitations des transputers, jusqu'au modèle T800 inclus.

4.2.1.1 Support pour les applications parallèles

Le transputer est, dans la philosophie de sa conception initiale, une brique de base pour la construction de machines parallèles sans requérir tout l'ensemble de composants matériels nécessaire aux processeurs classiques : un transputer est un **trans**istor com**puter** et offre, par cette intégration, un rapport performance/coût financier très compétitif.

Cependant, les réseaux de transputers ne se programment pas de la même manière qu'une seule unité. La répartition de programmes sur un réseau est limitée par le nombre de canaux de communication entre processeurs. Chaque transputer disposant de quatre liens, et donc de huit canaux, huit processus au maximum peuvent communiquer avec d'autres processeurs voisins du réseau à un instant donné.

Le modèle CSP, qui est le support théorique du langage occam, préconise le parallélisme et la communication comme paradigmes de construction de programmes, dont le degré de parallélisme logique et le nombre de canaux de communication sont élevés et virtuellement illimités. La contrainte du nombre de canaux entre processeurs limite sévèrement les performances attendues par l'utilisation transparente de plusieurs processeurs en réseau.

D'un autre côté, la correction théorique prouvable des programmes

écrits en occam conduit les concepteurs du transputer, prisonniers d'un modèle formel, à négliger le support d'observation, d'analyse et de mise au point de programmes. Il est par exemple impossible d'observer proprement l'état d'un processus ou d'un canal de communication, d'analyser le comportement d'un processus pour déterminer les ressources qu'il utilise, ou encore de traiter les conditions exceptionnelles et autres erreurs, et encore moins de tracer l'exécution d'un processus.

Cette naïveté initiale ferme la porte au support de nombreux langages de programmation et au bénéfice de la plus grande partie des programmes et d'outils systèmes du parc existant. Le transputer est perçu comme une machine langage, dédiée à occam sous TDS et INMOS se rend compte aujourd'hui de l'intérêt du marché d'autres langages et du support aux systèmes d'exploitation.

4.2.1.2 Support d'exécution parallèle

Le transputer supporte directement un modèle de processus à deux priorités et accomplit les fonctions d'ordonnancement. Il est très difficile de modifier la politique d'ordonnancement, et même d'en obtenir une variation. Les opérations de manipulation des files de processus n'offrent aucune garantie de cohérence, par exemple on ne peut interrompre un processus pour en redémarrer un autre. L'état d'un processus n'existe que pendant qu'il s'exécute, il n'y a pas d'administration globale des processus.

Il reste donc bien peu de marge pour une politique logicielle d'ordonnancement et pour la définition d'une abstraction de processus avec les opérateurs associés. D'autant plus que ces politiques ne pourraient être instaurées faute de l'autorité que confère un mode privilégié auquel seraient réservées les instructions d'administration.

A contrario, cette simplicité est aussi un intérêt très important du transputer, qui intègre un certain nombre de fonctionnalités traditionnellent implantées dans les systèmes d'exploitation. Il peut ainsi être utilisé sans recours à un système d'exploitation pour certaines applications spécifiques, de traitement d'image par exemple.

L'absence de mode privilégié n'est que l'un des aspects de l'absence totale de protection. Il n'y a pas de restriction d'accès à la mémoire et pas de comptabilité de zones mémoire par processus. Enfin, aucun support n'est prévu pour l'implantation d'une mémoire virtuelle.

4.2.1.3 Support système de bas niveau

Le débit des liens est faible pour une puissance de calcul annoncée à 10 Mips. Pour un débit maximal théorique de 10 Mbits/s par lien, cela fait au total 9.40 Mo/s par processeur, ce qui correspond au

débit donné par la règle de 1 Mips par Mo/s. Cet équilibre est atteint pour un transputer n'accomplissant pas de fonction de routage. D'un point de vue théorique, le transputer ne devrait donc pas accomplir de routage de par ses possibilités réduites de communication.

La récupération des erreurs est globale, c'est-à-dire qu'elle demande, pour toute erreur, l'arrêt du processeur. On ne peut déterminer avec certitude le processus fautif ni reprendre l'exécution. Le processeur s'arrête donc dans un état pour lequel il est possible de lire sa mémoire, mais non d'analyser la trace de l'erreur. De plus, les liens de communication ne permettent pas non plus de récupérer une erreur.

En conclusion, l'absence des opérateurs fondamentaux au support de système d'exploitation ainsi que l'impossibilité d'observer l'état du transputer ont pour le moment interdit l'implantation de toutes les fonctionnalités ordinaires d'un système d'exploitation sur une machine à base de transputers.

Conscients de ce manque, INMOS a introduit une nouvelle famille de transputers et de périphériques spécialement conçus pour les systèmes embarqués, et supportant les systèmes parallèles et les noyaux temps réel.

4.2.2 Une nouvelle famille de transputers

La machine Supernode regroupe des transputers, un composant de commutation, et divers dispositifs de support système. La nouvelle famille introduit avec le processeur T9000 un composant de routage, le C104, et divers composants permettant d'assurer l'interface avec les processeurs des familles précédentes. Le Supernode résoud le problème de la communication entre processeurs par la reconfiguration. Routage et reconfiguration sont deux techniques, utilisées de façon complémentaires dans le projet Supernode II, permettant à des processeurs non voisins de communiquer.

Les performances actuelles des composants de reconfiguration, ainsi que le démontrent les réseaux Oméga utilisés dans le projet RP3, permettent largement de se passer de routage, mais le contrôle dynamique de la reconfiguration dégrade la performance du dispositif de façon importante. De plus, tous les protocoles de communication (voir le projet iWarp) ne se satisfont pas, pour des raisons de performances, de l'utilisation de cette technique.

Il apparaît que les deux techniques doivent être complémentaires. La famille de processeurs T9000 utilise le routage comme technique de communication. La configuration du réseau de communication est déterminée statiquement à l'initialisation de la machine. Au vu de la complexité de contrôle que nécessiterait une reconfiguration du réseau de communication logique, seules des configurations statiques

sont envisagées pour le moment. Cependant chaque C104 se reconfigure dynamiquement pour associer une sortie à chaque message à acheminer.

Les machines parallèles futures offriront très certainement, comme le propose aujourd'hui Supernode et d'autres machines, les deux techniques simultanément.

4.2.2.1 Support pour les applications parallèles

C'est dans ce domaine que le changement le plus important intervient. Chacun des quatre liens du transputer supporte au plus huit canaux point à point uni-directionnels entre processeurs. Dans la nouvelle famille, chaque canal physique est multiplexé entre des "canaux virtuels" logés en mémoire.

Les canaux virtuels sont chacun représentés par une structure dans la mémoire de chacun des processeurs voisins et se partagent l'accès au canal physique par un ensemble d'instructions de communication identiques aux anciennes instructions. Le nombre de canaux virtuels entre deux processeurs voisins est potentiellement infini, comme pour les canaux internes à un processeur. Comme ces derniers, il est limité par la quantité de mémoire que l'on réserve à cet effet, le protocole de communication supportant un adressage sur un nombre d'octets extensible.

Cette technique permet le partage de canaux physiques entre processeurs *directement voisins*, mais ne résoud pas le problème de la communication transparente à travers tout le réseau. Un composant séparé, le C104, remplit la double fonction d'établissement de communications entre processeurs ou autres C104, et de multiplexage de canaux physiques en canaux virtuels.

Les T9000 et C104 peuvent être interconnectés suivant une topologie quelconque, chaque C104 permettant de relier 16 paires de liens bi-directionnels. Chaque T9000 possède 4 liens de communication, supportant chacun un débit annoncé à 10 Moctets/s, pour une puissance de 150 Mips. On obtient alors un débit total de communication de 80 Mo/s pour une puissance de calcul de 150 Mips.

Cependant, les processeurs ne sont plus impliqués dans les tâches de routage, et donc la communication n'altère pas la puissance de calcul. D'autre part, la mise en cascade de composants de communication C104 n'altère pas de façon importante le débit de bout en bout. INMOS annonce, pour une topologie en hypercube, une variation du délai de communication de $\frac{28}{63}$ pour une variation de diamètre de $\frac{27}{63}$, doublé ($\times 2.33$) au passage de 64 à 16384 processeurs. La famille T9000/C104 se place donc en bonne position pour le rapport de la puissance de calcul au débit de communication.

Les langages C, Fortran et Ada sont explicitement supportés en plus

du langage occam. Les états des processus et des canaux sont désormais mieux observables. Des instructions d'analyse et de mise au point de programmes sont disponibles.

Après observation des besoins réels des applications parallèles, le protocole de communication par échange synchrone de messages entre deux processus s'avère insuffisant. Un protocole de communication, construit à partir de ce protocole élémentaire, est tellement souvent utilisé que INMOS décide d'en faire un protocole directement supporté par le matériel. Il s'agit du protocole entre plusieurs émetteurs et un seul récepteur distribués dans le réseau. Les émetteurs sont mis en attente jusqu'à ce que le récepteur puisse accepter leur message. Ce protocole permet l'implantation de services systèmes.

On voit que le constructeur a tenu compte des critiques quant au support pour les applications parallèles. Cependant, le support système offert, notamment pour la gestion mémoire, est bien loin des supports disponibles sur d'autres processeurs plus classiques. D'autre part, la complexité du contrôle des composants a cru de façon importante, et la phase d'initialisation d'une machine est désormais hors de portée du programmeur non expert. De ce fait, la puissance nouvellement disponible peut être difficile à exploiter.

4.2.2.2 Support d'exécution parallèle

La famille T9000 introduit plusieurs types de processus pour implanter un mode maître et en même temps maintenir la compatibilité avec les anciens transputers. Les opérations de gestion des processus sont mieux définies et se prêtent à l'implantation de politiques différentes à partir de mécanismes de base.

Le mode maître permet à un processus d'en contrôler d'autres, dits esclaves, offrant ainsi un support aux systèmes d'exploitation pour appliquer dynamiquement leurs politiques propres lors des interactions des processus avec leur environnement. En effet, l'espace d'adressage des processus esclaves est surveillé et tout comportement illégal récupérable est rapporté, sous la forme d'une exception, au processus maître qui a la responsabilité de corriger l'anomalie.

Le T9000 implante donc un mécanisme d'exception basé sur la notion de processus maître, processus esclave et de traitant d'exception ("exception handler"). Ce mécanisme, en permettant la co-existence de plusieurs "maîtres" et en associant chaque processus "esclave" à un maître spécifique, permet de faire cohabiter plusieurs supports d'exécution, correspondant éventuellement à des politiques différentes.

Cependant, les processus maîtres ne sont pas protégés les uns par rapport aux autres, et certaines erreurs sont considérées comme fatales alors qu'elles devraient logiquement n'être que des exceptions.

4.2.2.3 Support système de bas niveau

Un gros effort a été porté sur la communication. Cependant, la puissance de traitement a proportionnellement plus augmenté que le débit des liens, soit un rapport de $\frac{150\ Mips}{80\ Mo/s}$ par processeur. Cette dissymétrie est compensée par la flaiblesse de l'incrément d'expansion pour le passage de petits réseaux (environ 64 processeurs) à des réseaux à plusieurs milliers de processeurs.

Un effort appréciable a été porté sur la récupération et le traitement des erreurs. Il est désormais possible, non pas de reprendre une instruction ayant causé une exception, mais pour un processus maître d'exécuter une instruction pour le compte de l'un de ses processus esclaves.

Certaines erreurs sur les canaux virtuels peuvent être récupérées, mais d'autres, logiquement sans conséquence pour les autres processus, sont considérées comme fatales pour tout le processeur.

4.3 Conclusion

L'idée du Supernode, construire une machine de forte puissance à moindre coût à partir de composants facilement ajustables, est un succès indéniable. La communication est un point crucial pour de telles machines, et doit être, autant que possible, intégrée en matériel pour des raisons d'efficacité.

Cependant, le choix d'une topologie reste une question essentielle pour les machines parallèles. Le reconfiguration dynamique est séduisante mais introduit un ensemble de problèmes liés au contrôle des dispositifs matériels de reconfiguration.

Les machines de la famille T9000, par une amélioration importante des débits de communication et des temps de routage, essaient de pallier au problème du choix de la topologie initiale. Le temps de traversée des composants de routage reste cependant une fonction linéaire du nombre de composants traversés.

Enfin, l'intégration de la communication en matériel mène à des points de congestion du réseau, qui ne sont pas faciles à éliminer dynamiquement avec une technique d'acheminement de messages intégrée. INMOS propose une adaptation de "l'Universal Routing" proposé par Valliant [Valliant82], qui demande encore des efforts de recherche.

Partie II

Parx : concepts et mise en œuvre

Chapitre 5

Motivations et Objectifs de Parx

La première partie nous a permis de mettre en évidence deux aspects importants de la programmation parallèle. Le premier est le support système nécessaire aux applications parallèles, domaine encore largement à explorer. Le second est l'ensemble des caractéristiques des architectures de machines utilisables par un système d'exploitation. Tous deux posent un ensemble de problèmes nouveaux importants auxquels les systèmes d'exploitation classiques n'apportent pas de solution.

Le système d'exploitation se trouve en fait devoir réaliser une double fonction. Il doit offrir un environnement d'exécution sur lequel puissent se projeter élégamment, c'est-à-dire sans être trop déformés, et efficacement les modèles de programmation des applications et des langages. La facilité de cette projection mesure l'adéquation réciproque des modèles d'exécution et de programmation et la simplicité d'utilisation de l'interface du système d'exploitation.

Une seconde fonction est l'exploitation de la machine physique afin d'en proposer une utilisation plus simple, et si possible tout aussi efficace.

Les deux objectifs de simplicité et d'efficacité semblent contradictoires et de nombreux systèmes d'exploitation ont fait le choix de dissimuler la complexité du parallélisme de la machine physique à l'interface de la machine virtuelle. Ceci est le sujet d'une controverse quant au niveau de complexité que sont capables de maîtriser les programmeurs, et donc de la complexité de la machine virtuelle. Les utilisateurs du noyau de système Parx sont des programmes systèmes, réalisant des services systèmes (serveurs de fichiers ou d'objets).

La première section donne un aperçu général de l'organisation du système. Nous présentons ensuite les objectifs et les motivations de Parx. Cette section explique pourquoi nous avons décidé une ap-

proche nouvelle pour les systèmes parallèles. Nous expliquons ensuite la motivation d'un nouveau modèle de processus. Nous terminons par la problématique du choix d'un modèle de processus.

5.1 Aperçu général du système

Le noyau du système Parx a été partiellement développé dans le cadre du projet Esprit Supernode II. Il constitue l'architecture de base du système d'exploitation PAROS, qui est un résultat de Supernode II. Le noyau de Parx a aussi été adopté pour PAROS. L'aspect communications, notamment les fonctions de reconfiguration et de routage, a été conçu et développé au sein de l'équipe et est décrit dans les rapports de thèse [Waille91] et [Gonza91]. Nous avons réalisé le noyau de communication et la gestion des processus légers. Le restant du modèle de processus a été réalisé par nos partenaires, ainsi que le développement des sous-systèmes PCTE et d'applications diverses.

5.1.1 Organisation

Nous avons vu que la classe des machines parallèles regroupait plusieurs architectures différentes. Un programme parallèle nécessite un support d'exécution et un contrôle spécifiques et cohérents. Pour cela, la machine virtuelle que lui propose le système doit être réellement parallèle, avec un support qui lui soit adapté. C'est pourquoi nous considérons que l'architecture physique doit être capable de faciliter la production de cette machine virtuelle. La figure 5.1 donne une vue générale des couches du système.

Une première couche de logiciel étend les possibilités du matériel et nous la dénommons "**extension matériel**". Elle contrôle le matériel et cache, dans la mesure du possible, ses spécificités. Elle permet le découpage dynamique de la machine physique en multiples machines virtuelles : les clusters, et réalise la communication de bas niveau entre tous les processeurs d'une machine virtuelle et entre machines. Elle a aussi pour fonction de fournir le flot d'exécution (ou processus léger) sur le processeur 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. Une particularité de ce noyau est qu'il permet de décomposer la machine en sous-systèmes. Chacun d'eux peut définir des politiques qui lui sont propres, à partir des mécanismes de base offerts par le noyau.

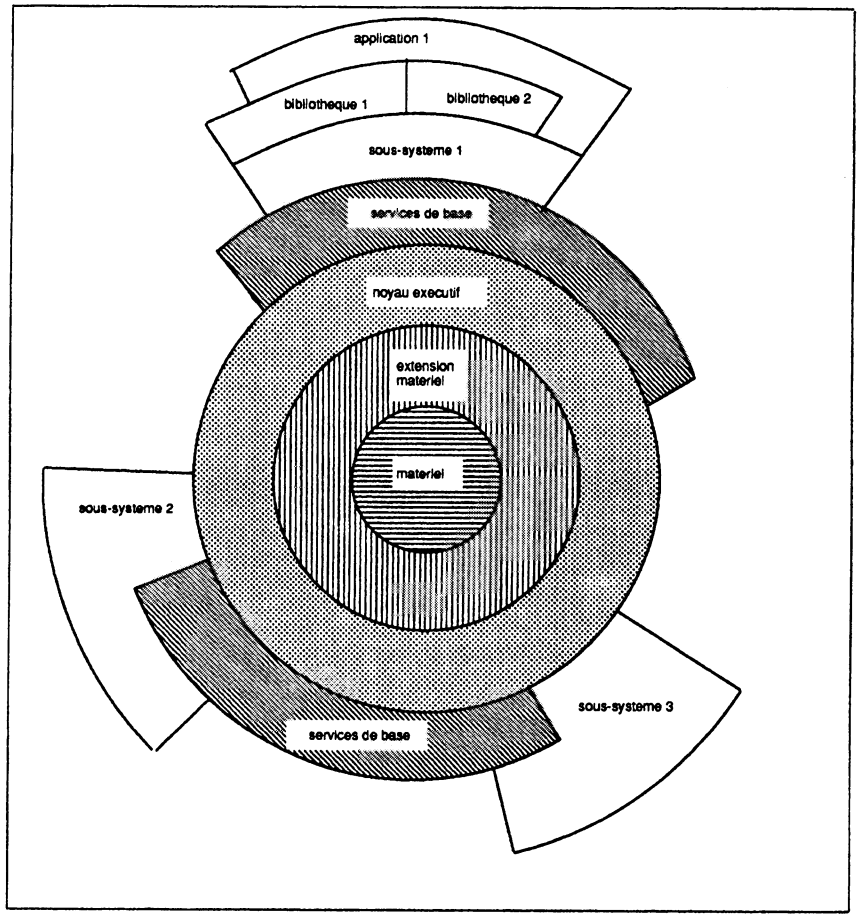


Figure 5.1: Aperçu général du système

Au dessus du noyau se trouvent les **services de base**, notamment pour accéder à des fichiers ou pour charger 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. Les sous-systèmes Portable Common Tools Environment (PCTE) et X/OPEN sont en cours d'implantation dans le cadre du projet ESPRIT Supernode II.

Chaque sous-système possède au moins un cluster. Il peut créér d'autres clusters sur lesquels s'exécutent des programmes utilisateurs protégés, tout en conservant le contrôle de ces clusters. Le cluster du sous-système est maître, les clusters fils sont esclaves. Ce mécanisme permet, par récurrence, de construire d'autres sous-systèmes.

Chaque sous-système met en œuvre un ensemble de politiques propres, à partir des mécanismes fournis par le noyau. Chacun offre à ses utilisateurs une vue particulière de la machine, certains celle d'une machine Unix standard, d'autres celle d'une machine numérique etc.

Enfin, les applications s'exécutent par dessus un sous-système parti-

culier.

5.1.2 Type de système et applications visées

Comme beaucoup de machines parallèles, les prototypes actuels fonctionnent avec une machine hôte qui assure l'initialisation de la machine cible, souvent encore les entrées-sorties disque et le système de fichiers, et les entrées-sorties terminal.

L'évolution prévue est vers une gamme de machines allant de la station de travail dotée d'un accélérateur de calcul, au gros serveur de calcul, système "stand-alone" autonome. L'accent est porté sur la flexibilité de la conception. Il ne s'agit pas, cependant, de réaliser un gros ordinateur supportant beaucoup d'usagers interactifs, mais plutôt sur un nombre moyen d'utilisateurs (quelques dizaines) effectuant des travaux nécessitant des puissances de calcul importantes. Chaque programme parallèle nécessite une machine virtuelle de forte puissance.

Il est à noter qu'un sous-système est supporté comme un ou plusieurs programmes parallèles, qui peuvent à leur tour fournir un support à de multiples utilisateurs fortement interactifs. Ceci sera illustré dans la section consacrée au modèle de processus.

Les applications visées sont de grosses consommatrices de puissance de calcul et de communication. Il s'agit notamment d'algorithmique numérique et parallèle, d'applications de simulation et de bases de données. Elles s'expriment dans des langages dont le support pour le parallélisme a été étudié au chapitre 2.

Elles s'exécutent dans l'environnement de sous-systèmes. Les composantes d'un sous-système dédié, à la différence de PCTE et X/OPEN, au support d'applications parallèles, sont à l'étude ou à l'état de prototypes dans le projet Parx. Ils devront être intégrés dans le futur. Ils couvrent la conception, le développement, la compilation, le support "*run time*" et la mise au point de programmes parallèles.

5.2 Objectifs de Parx

Nous précisons la définition de quelques termes dans le contexte de cette étude. Le **modèle de programmation** d'un langage, que [BSTan89] appelle tout simplement *modèle* d'un langage, est défini par un ensemble d'*entités* —ou objets— et d'*opérations* sur ces objets. Il correspond au jeu d'instructions d'une machine interprétant directement le langage. Nous appelerons cette machine, cible du programmeur, machine abstraite. Dans les langages comme CSP

[Hoare78] ou occam [Inmos84b], le modèle de programmation est constitué d'un ensemble de processus communiquant, se synchronisant uniquement par échanges de messages.

Le **modèle d'exécution** offert par un système d'exploitation est défini par l'ensemble des entités construites par le système, et des opérations sur ces entités. Les entités peuvent être des objets, les opérations étant l'envoi de messages déclenchant l'exécution de méthodes [Bal87]. Elles peuvent aussi être des processus manipulant des fichiers ou d'autres processus comme c'est le cas du système Unix [Bach86]. Il correspond à la sémantique des entités et des opérateurs fournis par le système d'exploitation, qui sont eux-mêmes une virtualisation des mécanismes offerts par le matériel. Nous appelerons cette machine, construite par le système d'exploitation, machine virtuelle. Le concept de machine virtuelle, introduit par [MeSea70] avec CP/67, puis popularisé par le système VM/370, tend généralement à désigner l'interface de programmation système. Certains auteurs parlent de *processeur virtuel* [Black90].

5.2.1 Support de modèles de programmation

5.2.1.1 Etat de l'art

Le projet Parx part d'un constat simple : le développement des machines parallèles contraste fortement avec l'absence évidente d'outils adaptés à leur exploitation. De nombreuses recherches sont menées dans cette voie mais la complexité des problèmes abordés conduit à l'adoption de solutions pratiques et généralement à court terme. L'utilisateur est en effet très peu assisté durant le développement d'une application parallèle et encore moins pour l'exécution de celle-ci.

L'aspect "langage pour la programmation parallèle" domine actuellement la recherche et voit l'éclosion de langages tels Linda [CaGel89a], Orca [BKT90] etc. La répartition des responsabilités entre le système d'exploitation et les environnements de programmation est très variable, et ces langages, comme nous l'avons vu en introduction du chapitre 2, endossent pour la plupart une partie des attributions traditionnellement dévolues aux systèmes d'exploitation.

Cet état de faits est probablement dû à la difficulté de développer des systèmes d'exploitation pour machines parallèles. Citons tout de même Helios [Gar87], que nous décrivons en 9.2.3, Topexpress, Trollius [Braner88] et Idriss [King88] qui ont pour machines cibles des réseaux de transputers et sont des premières approches dans ce domaine. Nous parlerons de PEACE [WSP90] et EDS [IsBor90], qui sont d'autres systèmes parallèles, au chapitre 9.

L'aspect modèle d'exécution devrait cependant être complémentaire

de l'aspect modèle de programmation, et l'absence de système d'exploitation pour ces machines constitue à nos yeux un très sérieux handicap.

Il existe un double frein à ce développement : le manque d'expérience au niveau système pour les machines parallèles et la perception encore trop diffuse de ce qu'est l'exécution d'une application parallèle dans un tel système. Potentiellement, toute application est candidate à une exécution parallèle, soit pour un gain d'efficacité, soit parce qu'elle exhibe du parallélisme explicite. La multiplicité des supports système nécessaires à l'exécution de toutes les gammes d'applications apporte encore à la confusion quant aux modèles d'exécution parallèles.

Il faut rappeller que les systèmes d'exploitation aussi sont souvent des programmes parallèles. En effet, ils mènent "simultanément" plusieurs activités : contrôle de chaque périphérique, exécution de chaque programme utilisateur ou encore administration. Celles-ci correspondent généralement à des processus différents, ce qui est le cas dans Unix [Bach86] ou encore Minix [Tanen87].

Les systèmes d'exploitation utilisent le parallélisme à quatre fins :

1. disponibilité à un plus grand nombre de programmes,
2. accélération du traitement,
3. résistance aux pannes,
4. utilisation de processeurs spécialisés.

5.2.1.2 Approche adoptée dans Parx

Il est largement reconnu que le concepteur d'une application dispose d'un ensemble d'informations, qui parfois n'apparaissent ni explicitement, ni implicitement dans son programme, quant à la façon de faire efficacement usage du parallélisme. C'est le rôle de l'environnement de programmation et d'exécution de demander, en proposant des outils d'aide à l'écriture, et parfois d'extraire automatiquement des programmes sources, cette information afin de l'utiliser au mieux.

Cependant, le concepteur de systèmes d'exploitation est à son tour le mieux placé pour déterminer comment doivent être utilisées les ressources qu'il administre pour un meilleur rendement. Il restera donc dans les attributions du système d'exploitation de gérer les ressources.

Il y a par conséquent un compromis à déterminer entre les besoins exprimés par le programmeur à travers l'environnement de programmation, et l'utilisation optimale du système d'exploitation déterminée par ses concepteurs. Le projet Parx mène de front l'étude de ces deux aspects des systèmes parallèles.

Un réseau utilisateur sera donc constitué de processeurs virtuels qui peuvent correspondre à des processeurs physiques partagés entre plu-

sieurs applications, cette correspondance évoluant éventuellement au fil du temps. De même qu'un seul processeur peut correspondre à plusieurs processeurs virtuels, une connexion physique supporte, en les multiplexant, plusieurs connexions du réseau virtuel.

De même, les services système seront représentés par un réseau dit "système". Le chargement de programmes, l'accès aux fichiers ou aux périphériques, la communication avec d'autres applications, seront conditionnés par l'existence d'une connexion avec le réseau système.

Ce schéma de programmation et d'exécution découle d'un principe adopté dans le projet Parx : l'environnement de programmation et le système doivent libérer le concepteur d'applications du problème de l'adéquation de son algorithme à la machine sur laquelle il l'exécutera. Un système qui obéit à ce critère est dit **massivement parallèle**, car le parallélisme offert à l'utilisateur n'est pas limité par les contraintes particulières de la machine. Nous pensons que l'accomplissement de cet objectif accélèrera l'adoption par la communauté scientifique, puis par le grand public, des systèmes parallèles.

5.2.2 Support Architectural

5.2.2.1 Etat de l'art

En fonction des facilités offertes par la machine et de l'usage qu'en fait le système d'exploitation, on retrouve deux types de systèmes parallèles.

Le premier type regroupe les systèmes bâtis sur une architecture homogène et un réseau de communication régulier, où l'architecture maître-esclave se retrouve sur chaque processeur. Ces systèmes sont construits sur le même motif qui peut se répéter par simple ajout de processeurs suivant le schéma du réseau de communication.

Beaucoup de machines sont construites selon ce schéma, mais peu de véritables systèmes d'exploitation le supportent. Les machines offrent beaucoup de processeurs, et le parallélisme massif est souvent mis en œuvre. L'utilisateur a toute la responsabilité de la détection et de la gestion du parallélisme, ainsi que du placement des constituants de son programme. Les services systèmes se limitent à ceux que l'on peut attendre d'une machine séquentielle, avec un mécanisme de désignation des processeurs et de placement de programmes sur ceux-ci.

Le second type comprend les machines à architecture hiérarchique. La machine a une topologie de base fixée, à laquelle correspond une organisation logicielle adaptée. Des systèmes plus importants sont construits en agençant de "petites" machines suivant le même réseau d'interconnexion. L'architecture maître-esclave se retrouve au sein de chaque composante de la machine.

De nombreuses machines suivent ce principe, mais bien peu de systèmes d'exploitation parallèles s'y sont adaptés. Bien que ce principe permette de mieux profiter de l'expérience acquise pour des machines monoprocesseur, les systèmes qui s'en réclament souffrent souvent des mêmes inconvénients que ceux de la classe précédente.

5.2.2.2 Approche adoptée dans Parx

Plusieurs auteurs se sont intéressés aux topologies des machines parallèles à divers points de vue. Une première approche est celle de la topologie la mieux adaptée à la résolution d'un problème particulier. Une seconde, brièvement décrite en section 3.2.2.1, est la classification des topologies en fonction de leurs qualités *a priori* pour la résolution d'un problème quelconque.

Une approche plus pragmatique est utilisée ici. L'expérience pratique de la programmation parallèle nous conduit à introduire un critère supplémentaire dans l'étude d'une topologie : la connectivité. En premier lieu, un programme utilisateur correspond à un sous-réseau de processeurs de connectivité adaptée. Nous parlerons de la connectivité d'un sous-réseau dit *utilisateur* par rapport à un autre dit *système*. En effet, toute application doit être chargée en mémoire principale ; en général à partir d'un support de mémoire secondaire, typiquement un disque. Ensuite, l'usage de flots d'entrées-sorties requiert la connexion au réseau système. Nous ne dissocierons donc jamais les programmes utilisateurs des services système qu'il faut leur fournir.

La recherche sur les systèmes mono-processeurs a conduit à distinguer les applications dont la performance est limitée par la puissance de l'unité de traitement (elles sont dites "*compute bound*"), de celles limitées par les unités de communication (elles sont dites "*I/O bound*"). Les systèmes d'exploitation administrent, souvent explicitement, ces deux classes d'applications par des algorithmes d'ordonnancement appropriés. Ce rappel montre que la connectivité avec le réseau système est un critère important d'efficacité.

Il est donc essentiel, dans une implantation de système parallèle, d'accorder beaucoup d'importance aux possibilités d'entrées-sorties sur des périphériques fréquemment sollicités, de type disque ou terminal.

5.3 Motivations d'un nouveau Modèle de Processus

Avec le succès des machines parallèles se pose le problème de leur programmation et du choix du type de système d'exploitation à adopter.

Comme nous l'avons rappelé lors de la définition des systèmes parallèles (voir section 1.1) et des systèmes massivement parallèles dans la section précédente, nous nous intéressons ici à un type de parallélisme et de pseudo-parallélisme que n'ont pas pris en compte les systèmes d'exploitation actuels.

Au niveau des systèmes d'exploitation pour le parallélisme, la recherche et les développements sont bien moins avancés que pour les langages parallèles. Le comportement à l'exécution des programmes parallèles est encore mal connu, bien que des unités de mesure soient proposées dans la recherche, par exemple un "processor working set" [Ghosal90], ou encore des modèles de comportement [Ferra90].

D'autre part, la conception et le développement d'un système d'exploitation sont une entreprise beaucoup plus longue et compliquée que ceux d'un langage de programmation. La contrainte de respecter un standard existant est aussi beaucoup plus forte pour le succès d'un système.

Les systèmes distribués forment la base des premiers travaux sur les systèmes parallèles, et offrent ainsi une alternative à l'approche purement langage jusque-là adoptée pour le parallélisme.

Le standard de fait, surtout dans le monde de la recherche, étant le système Unix, une large majorité de systèmes distribués ou parallèles sont bâtis avec une compatibilité Unix.

5.3.1 Approche système à partir du standard Unix

Le système Unix, très répandu dans le domaine universitaire et de plus en plus dans l'industrie, offre une forme limitée de pseudo-parallélisme et des primitives de communication au niveau de l'interface système. L'un des atouts d'Unix est son interpréteur de commandes, qui fournit à l'utilisateur des commandes directes pour créer des tubes ("pipes") de processus communiquants.

Au niveau système, l'on peut créer des graphes quelconques de processus, les arêtes en étant des tubes de communication uni-directionnelle. Cependant, les processus et les tubes Unix ne permettent qu'une exploitation très limitée du pseudo-parallélisme ou du parallélisme. En usage normal, les processus Unix utilisent longuement la puissance de calcul et communiquent relativement rarement.

Les communications doivent être longues (échanges d'importants volumes de données) et les correspondants ne passent pas rapidement d'un tube de communication à un autre. Elles sont interfacées par le système de fichiers, et soumises au mécanisme général d'accès aux fichiers.

Le modèle de processus Unix n'est pas adapté au parallélisme pour

plusieurs raisons :

1. la gestion de processus implique un surcoût important, notamment quant à la commutation de contexte : le processus Unix englobe un environnement d'entrées-sorties, la gestion d'une mémoire virtuelle (dans la large majorité des cas) et de ressources système. Afin de rentabiliser l'opération de commutation, il faut des durées subséquentes d'exécution relativement importantes (quelques centaines de millisecondes).

2. la primitive de création de processus opère par clonage du processus appelant. Ceci ne permet de créer qu'un seul processus à la fois, et laisse la charge de la création du réseau de communication au processus père, par un ensemble de primitives dont l'utilisation est complexe (pipe, dup, open, close).

   D'autre part, le nombre de tubes de communication est limité par processus (une vingtaine en général). On peut ainsi construire n'importe quel réseau de processus de degré maximal vingt, mais le procédé est long, fastidieux et coûteux pour des applications parallèles.

3. la création d'un processus est souvent suivie du remplacement de l'image mémoire du processus nouvellement créé par celle d'un autre programme (exec). Ce schéma est le seul moyen d'obtenir des processus exécutant des programmes différents, et son usage très important démontre une demande réelle de mécanismes plus appropriés pour la gestion du pseudo-parallélisme et de la communication.

4. les primitives de communication entre processus correspondent à un gros grain de parallélisme. De plus, le protocole de communication par tubes ne convient pas à la communication intensive à grain fin dans un modèle de processus parallèles communicants.

5.3.2 Pseudo-parallélisme sous Unix

Pour offrir un modèle de parallélisme au niveau utilisateur, différentes approches ont été explorées à partir d'Unix.

Une première méthode consiste à enrichir le modèle de processus par l'addition de bibliothèques. Une seconde approche procède par la modification de la sémantique du processus Unix et fournit de nouveaux appels systèmes pour supporter les modifications. Des processus dits "légers", des "tâches" ou encore des "threads" ont été introduits, ainsi que des primitives de communication leur correspondant.

La première approche est la plus commune et requiert relativement peu d'efforts de conception. Elle permet, tout en conservant une base

traditionnelle, de rajouter une couche de logiciel modifiant l'interface sous-jacente.

Les inconvénients découlent de cette facilité apparente : l'environnement de processus légers étant bâti à partir d'un modèle de processus séquentiels "lourds", plusieurs primitives de base pour le pseudo-parallélisme et la communication sont incompatibles avec les concepts du support. Une grande partie de la puissance de la machine abstraite est perdue pour les processus légers, notamment les appels système bloquants.

La seconde approche demande des changements au noyau du système. On peut distinguer deux statégies d'implantation possibles, illustrées par les systèmes dits à micro-noyaux (Mach [TeRa87,Tevan87,Baron88] et Chorus[AHLR89]), et les systèmes plus monolithiques comme Symunix [ELS88b].

Mach

Il offre un modèle d'exécution différent d'Unix, permettant d'exploiter le pseudo-parallélisme et le parallélisme. Il distingue dans le processus Unix une partie environnement, comprenant notamment le contexte d'exécution et les ressources, et une partie flot de contrôle comprenant différentes activités s'exécutant dans le même contexte.

Il fournit une bonne approche pour supporter à la fois les applications traditionnelles Unix, sans usage de pseudo-parallélisme ou de parallélisme, et celles qui ne demandent que la parallélisation d'un seul algorithme, sans effets de bord sur son environnement d'exécution.

L'inconvénient est que dans un processus Unix contenant plusieurs flots de contrôle, il n'est pas possible de faire appel à plusieurs primitives de bibliothèque standard en parallèle. En effet, celles-ci peuvent effectuer des appels systèmes Unix dont la sémantique d'implantation demande que le contexte du processus n'évolue pas pendant leur traitement. Ceci est particulièrement vrai pour les entrées-sorties, ou les signaux.

Il est clair que la parallélisation d'une application doit s'accompagner de celle de l'environnement si celui-ci peut être appelé par différents flots d'exécution.

Symunix

Il conserve l'approche monolithique d'Unix. Les processus ont la même sémantique que ceux d'Unix, mais les goulots d'étranglement dûs aux sections critiques sont éliminés dans la mesure du possible. Ainsi, il offre des communications de faible coût entre processus, et introduit une création de processus par clonage multiple. De plus,

les processus sont regroupés par ensembles au sein desquels sont appliquées des politiques d'ordonnancement cohérentes.

Cette approche permet le portage d'applications composées d'un ensemble de processus de pleine sémantique Unix. Sans qu'il soit besoin de les paralléliser, on obtient de meilleures performances sous Symunix grâce à l'élimination des goulots d'étranglement, qui favorise le parallélisme et diminue le coût des communications entre processus. D'autre part, il n'est pas besoin de support particulier pour ces processus au niveau du noyau : ils peuvent effectuer des appels système en parallèle.

5.4 Problématique du choix d'un modèle de processus

Les progrès réalisés dans le domaine des systèmes d'exploitation, essentiellement distribués (Amoeba [Mullen90], Chorus [AGHR89], Mach [Accet86], V [Cher88]), multiprocesseurs (Nectar [ABCKSS89]), ou orientés objets (Amber [Chase89], Emerald [Black86], Guide [Frey91]), permettent aujourd'hui de mieux comprendre le problème du choix d'un modèle de processus.

Les systèmes d'exploitation ont souvent, par le passé, regroupé au sein d'une seule entité les fonctions de traitement, mettant en œuvre des flots d'exécution d'instructions, et celles de conservation et d'accès aux données. Nous l'avons vu à la section 3.1, ces deux fonctions peuvent être clairement séparées. Nous précisons cette séparation que nous illustrons en nous appuyant sur un système réparti à objets, Guide, développé au sein de l'unité mixte de recherche Bull-IMAG à Grenoble.

5.4.1 Flots d'exécution

Deux niveaux de programmation se distinguent : langage et système d'exploitation. Chacun définit un ou plusieurs flots d'exécution, et il se pose le problème de la projection des flots d'exécution du langage sur ceux offerts par l'interface système.

Guide résoud ce problème en identifiant les flots du langage Guide [Frey91] et ceux du système. Le flot d'exécution est appelé "activité", et s'exécute dans un "domaine" qui est une machine virtuelle pouvant s'étendre sur plusieurs sites. Un domaine peut comporter un nombre indéterminé d'activités. On a ainsi plusieurs processeurs virtuels par domaine, en considérant que chaque site fournit un processeur virtuel. Les processeurs virtuels s'exécutent en parallèle, les activités d'un même processeur virtuel en pseudo-parallèle.

Différentes combinaisons se rencontrent dans les systèmes d'exploitation. La possibilité d'exécution parallèle n'est offerte que depuis une huitaine d'années, et généralement sous la forme de primitives permettant à des programmes complètement étrangers de collaborer : lancement de plusieurs programmes à la fois (*fork* multiple [ELS88a]), échange de messages etc.

Guide offre un environnement cohérent aux programmes parallèles : le domaine. Il permet à un programme de s'exécuter en parallèle sur différents sites. Une activité peut engendrer l'exécution synchrone d'une activité sur un autre site. Elle est un flot d'exécution synchrone multi-sites.

Du fait que le domaine représente un espace d'adressage commun à ses activités, la "visite" d'une activité à un autre site met en jeu des opérations de liaison complexes. Le grain de pseudo-parallélisme, l'activité, demande donc une gestion plus importante que celle des processus légers plus classiques. Le grain est moins important que celui impliqué par un processus Unix par exemple, mais beaucoup plus que celui d'un processus léger.

5.4.2 Accès, conservation des données

Un flot d'exécution se déroule dans un espace d'adressage. Deux cas peuvent se présenter : les données peuvent être directement accessibles dans l'espace d'adressage, ou bien demander un protocole d'accès indirect.

Dans le premier cas, on peut avoir des mécanismes de liaison dynamique, qui peuvent impliquer une gestion plus lourde des espaces d'adressage. Le tableau 5.1 présente les combinaisons entre flot d'exécution et espace d'adressage. Il ressemble à la classification de Kuck pour les architectures de machines. Nous voyons que, en fonction des objectifs des systèmes d'exploitation, différents choix sont possibles.

	espace d'adressage unique	espaces d'adressage multiples
flot unique	un flot par espace (programmes classiques, objets actifs)	un flot, plusieurs espaces (migration, RPC)
flots multiples	plusieurs flots par espace (pseudo-parallélisme)	plusieurs flots, plusieurs espaces (parallélisme)

Table 5.1: Combinaisons flots d'exécution, espaces d'adressage

protocole d'accès aux données	style de programmation
procédural	objets
échanges de messages	client-serveur

Table 5.2: Accès aux données

Le modèle un flot, un espace, peut être utilisé dans les systèmes

à objets actifs. Il est le plus souvent utilisé dans les systèmes classiques, dans lesquels le processus regroupe le flot d'exécution, l'espace d'adressage, et toutes les fonctionnalités du système. On obtient alors un modèle de processus lourd, en particulier en environnement distribué et plus encore en environnement parallèle. La tendance des systèmes d'exploitation est de s'éloigner de ce monolithisme (Mach [Teva87b], Chorus [AHLR89]). Ses inconvénients sont sa lourdeur et le manque de flexibilité dans l'utilisation des fonctionnalités, engoncées dans un cadre unique omniprésent.

Un flot, plusieurs espaces d'adressage correspond à des systèmes distribués à processus pouvant migrer (Emerald [Jul88]), ou effectuer des appels de procédure à distance. La migration ne s'accompagne pas de celle de l'espace d'adressage, et son coût en est ainsi diminué. Il reste que cette opération, du fait de l'héritage d'environnement qu'elle implique, est d'un coût beaucoup plus important que l'activation d'un flot de données dans un espace d'adressage extérieur. Elle ne peut donc s'appliquer que pour des blocs d'instructions plus importants.

Plusieurs flots, un seul espace correspond à du pseudo-parallélisme. Ce modèle (Mach, Chorus) a succédé aux systèmes monolithiques. Il découple les flots d'exécution de l'espace d'adressage, permettant ainsi de mettre en œuvre plusieurs activités se déroulant dans le même espace. Ceci est particulièrement utile pour les interactions bloquantes avec l'environnement. Il peut s'agir d'écritures bloquantes ou de l'acquisition de données provenant de différentes sources. Il permet une écriture plus "naturelle" d'algorithmes qui mettent en jeu simultanément plusieurs activités : exploration de plusieurs solutions par exemple. Cependant, il ne permet pas de profiter du parallélisme effectif d'une machine (excepté dans les systèmes multi-processeurs à mémoire partagée).

Enfin, le modèle plusieurs flots, plusieurs espaces permet d'exhiber le parallélisme à travers la multiplicité des espaces d'adressage. Il est ainsi visible (le modèle précédent peut offrir un parallélisme transparent), et contrôlable au mieux des intérêts d'un utilisateur. Il présente les avantages du modèle précédent, tout en éliminant ses inconvénients. Cependant, en rendant orthogonaux les concepts traditionnellement liés aux processus, il accroît la complexité du contrôle à mettre en œuvre. Un effort important doit être fait pour faciliter aux utilisateurs la compréhension du modèle, et fournir des interfaces d'utilisation simples mais confortables. Ce rapport est une contribution à cet effort.

Le modèle de conservation de données adopté dans Guide pour ses premières réalisations utilise des objets passifs mono-sites. Chaque domaine correspond à un seul espace d'adressage qui peut être distribué. Guide se situe entre les modèles "un flot, plusieurs espaces" et "plusieurs flots, un espace". Il permet de créer plusieurs activités s'exécutant dans un seul espace d'adressage, mais l'espace d'adres-

sage est distribué sur plusieurs sites. Le mécanisme d'invocation des objets permet, ceux-ci étant mono-site, de mettre en œuvre la distribution de l'espace d'adressage. Le modèle sous-jacent est un appel de procédure sur un autre espace d'adressage.

Les objets sont liés dans cet espace au fur et à mesure qu'ils sont utilisés. La localisation des objets est transparente à l'utilisateur. Un tel schéma a l'avantage d'abstraire l'utilisateur du problème de la localisation des objets. Cependant, sa mise en œuvre suppose un certain coût dans le mécanisme de désignation et dans la gestion des noms dans un espace d'adressage. Ce coût n'est pas significatif, par rapport aux facilités offertes par une désignation homogène, pour les applications visées par le système. Ce sont des applications coopératives, mettant en jeu des interactions nombreuses et un partage d'information complexe entre plusieurs utilisateurs : les grains de parallélisme et de communication sont donc plus gros que ceux envisagés dans Parx. Le parallélisme physique est caché par cette homogénéité. Guide offre d'autre part des moyens d'accéder à la localisation physique des objets et de l'influencer.

5.4.3 Bilan

Nous voyons que le choix d'un modèle de processus dépend essentiellement du type des applications visées et de la vue cohérente que l'on veut donner à l'ensemble du système. Plusieurs approches sont menées par différents projets, qui ont leurs mérites et limitations respectifs.

Parx s'intéresse aux applications à haut degré de parallélisme, et par conséquent doit mettre en place un modèle et des techniques adaptées à ces environnements.

Précisons maintenant la notion de grain de parallélisme. Nous distingons trois mesures. Les deux premières sont relatives à l'exécution d'un programme, la dernière aux caractéristiques d'un nœud. Le grain d'exécution d'un programme est le rapport du nombre moyen de blocs d'instructions que l'on peut exécuter en parallèle au nombre moyen d'instructions d'un bloc exécuté en parallèle. Le grain de communication d'un programme est le rapport de la fréquence des communications à la quantité moyenne d'information échangée lors d'une communication.

$$\textit{grain d' exécution} = \frac{\textit{nombre moyen de blocs exécutables en parallèle}}{\textit{nombre moyen d'instructions par bloc parallèle}}$$

$$\textit{grain de communication} = \frac{\textit{fréquence des communications}}{\textit{volume moyen d'une communication}}$$

$$\textit{équilibre d'un noeud} = \frac{\textit{puissance du processeur de traitement}}{\textit{puissance du processeur de communication}}$$

La troisième mesure prend en compte les puissances comparées de

calcul et de communication. En fonction de ces paramètres, chaque nœud peut atteindre un grain de maximal de communication : c'est le rapport de la puissance du processeur de traitement à celle du processeur de communication.

Nous avons étudié ce dernier rapport à la section 3.3.2. Rappelons que pour les transputers T800, les nouveaux processeurs T9000 d'INMOS et le processeur d'iWarp, les rapports sont respectivement de $\frac{10-15}{10}$, $\frac{150}{80}$ et de $\frac{20}{320}$ Mips par Mo/s. Un rapport supérieur à l'unité indique la meilleure performance en communication, et invite à distribuer des programmes communiquant beaucoup. Ce rapport, comme nous l'avons déjà indiqué en 3.3.2, doit être modulé en fonction de la charge de routage que doit accomplir le processeur. Le transputer perd une partie importante de sa puissance de traitement en routage. Dans le cas du T9000, cette perte est nulle, le processeur n'accomplissant pas de fonction de routage. Nous avons présenté la famille de processeurs T9000 en 4.2.2. Leur débit peut cependant être ralenti si les composants de routage sont en congestion. Enfin, iWarp dispose, pour chaque processeur de calcul, d'un processeur de communication fonctionnant en parallèle et accomplissant les fonctionnalités de routage. La bande passante de communication est donc une fonction de sa charge. Peu de mesures ont été effectuées quant aux temps moyens de calcul et de communication pour des applications parallèles.

Chapitre 6

Le modèle de processus

Nous présentons dans ce chapitre le Modèle de Processus de Parx.

Nous introduisons tout d'abord la notion de Cluster, qui sert au découpage d'une machine physique. Dans la seconde section, le modèle de Parx est détaillé, avec les principales opérations qui s'y rapportent. Les primitives décrites sont tirées d'un document de travail ESPRIT Supernode II, leurs prototypes sont donnés en C ANSI.

La spécification finale, développée dans le cadre du projet Supernode II, est largement basée sur une spécification préliminaire que nous avions rédigée. Bien entendu, cette spécification est le fruit de nombreuses discussions avec nos collègues de recherche, et les commentaires de nos partenaires du projet ont largement contribué à son amélioration. Un prototype du modèle de processus a été réalisé dans le cadre de ce projet.

6.1 Les Clusters de Parx

L'étude menée sur les machines parallèles nous a permis de mettre en évidence la nécessité du contrôle et du support système. La machine n'est pas seulement un ensemble de processeurs dont peuvent disposer les applications à volonté, mais offre à chaque application une machine parallèle correspondant autant que possible à ses exigences de puissance de traitement et de communication, avec un support d'exploitation comparable aux machines mono-processeur classiques. Chaque grappe de processeurs, appelée **cluster**, allouée à une application correspond à une partie utilisateur s'exécutant en mode esclave sur un processeur classique. Un cluster système correspond au mode maître. A son tour, l'utilisateur peut manipuler le cluster comme une machine complète. L'objectif affiché est de rétrocéder le maximum de la puissance matérielle disponible aux applications.

6.1.1 Le Cluster

C'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. 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.

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 en deçà 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 de 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.

Nous avons vu que les clusters utilisateurs correspondaient 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 le flot d'exécution ayant provoqué l'appel est suspendu. 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.

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 le système de protection des entités du système.

Le concept de cluster découle de l'étude des chapitres 2 et 3, qui font apparaître la nécessité de subdiviser une grosse machine en sous-réseaux plus petits, auxquels puissent s'appliquer un contrôle plus souple et plus efficace. Il est illustré par la figure 6.1.

6.1.2 Interface de manipulation

Chaque objet de Parx est désigné à l'aide d'une capacité renfermant les droits de son possesseur sur l'objet.

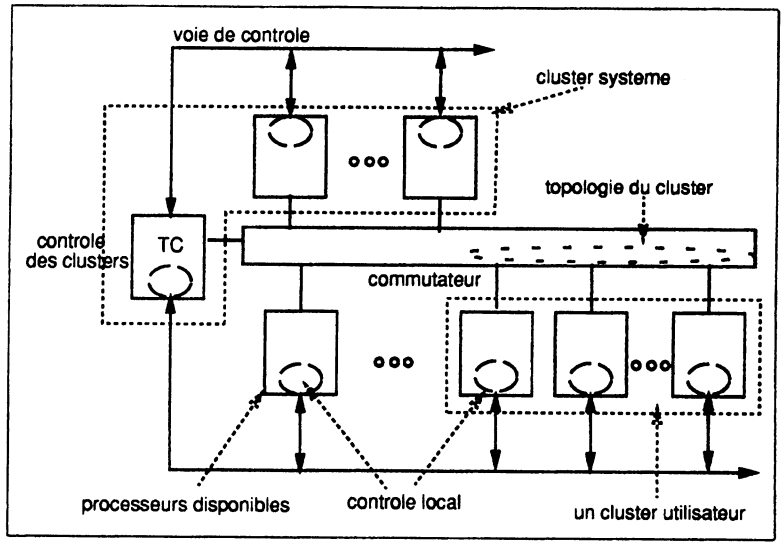


Figure 6.1: Illustration d'un cluster dans un Supernode

6.1.2.1 Création

Un cluster peut être créé par la primitive :

```
t_error cluster_create ( t_cluster * id_cluster,
                         t_processor_desc processor_type, int nombre_proces-
                         seurs_min, int * nombre_processeurs, int quota,
                         t_processor_array id_processeurs[]
                       )
```

Un cluster est une machine virtuelle formée d'un certain nombre de processeurs. Le paramètre id_cluster sert à retourner une capacité désignant le cluster créé. processeur_type décrit les caractéristiques du processeur "type" du cluster. Il s'agit de son type (T800, T9000 etc. dans le cas des transputers), de la taille de sa mémoire etc. nombre_processeurs_min est la borne inférieure du nombre de processeurs à allouer, en deçà de laquelle la création ne doit pas se faire. nombre_processeurs est le nombre de processeurs idéalement souhaité. Au retour de la primitive, il indique le nombre de processeurs effectivement alloué au cluster. quota est une borne supérieure au nombre de processeurs que pourra au total réclamer le cluster ou sa descendance. Chaque processeur du cluster est désigné par un identificateur qui est retourné dans le tableau id_processeurs des processeurs alloués.

Afin de permettre une administration des ressources "processeurs" allouées aux clusters, chacun d'eux possède un quota de processeurs qu'il peut simultanément avoir alloué pour ses besoins propres ou ceux de sa descendance. Ainsi, les clusters sont soumis à une relation de filiation arborescente. Cette relation va servir de support objectif aux mécanismes maître-esclave mis en œuvre par les sous-systèmes, que nous décrirons plus loin.

6.1.2.2 Destruction

```
t_error cluster_delete ( t_cluster id_cluster
                       )
```

permet de détruire un cluster précédemment créé. Ceci ne préjuge en rien de l'exécution de programmes sur les processeurs du cluster. Le cluster n'effectue plus aucune activité visible. Le contenu des mémoires des processeurs est indéfini à la suite de cette opération. Les processeurs ainsi libérés peuvent être utilisés au gré du système d'exploitation.

Initialement, tous les processeurs libres sont dans un cluster géré par le système d'exploitation. De même que les créations de cluster puisent dans cette réserve de processeurs libres, les libérations l'alimentent.

6.1.2.3 Manipulation de clusters

Le nombre de processeurs d'un cluster peut croître ou décroître au cours de son existence.

```
t_error cluster_add ( t_cluster id_cluster, int mini, int maxi,
                      int * nb_processeurs, t_processor_desc proces-
                      seur_type, t_processor_array id_processeurs[]
                    )
```

permet d'ajouter à un cluster id_cluster au moins mini processeurs. maxi est le nombre réellement souhaité. Un nombre nb_processeurs de processeurs, correspondant à la description donnée par processeur_type, est effectivement alloué. Leur identification est retournée dans le tableau id_processeurs.

```
t_error cluster_remove ( t_cluster id_cluster, int nb_processeurs,
                         t_processor_array id_processeurs[]
                       )
```

permet de retrancher à un cluster id_cluster un nombre nb_processeurs de processeurs dont les identificateurs sont donnés dans le tableau id_processeurs.

Ces manipulations doivent se faire dans la limite du quota attribué à chaque cluster. Ce quota peut être changé par la primitive :

```
t_error cluster_resize ( t_cluster id_cluster, int nouveau_quota
                       )
```

Le cluster id_cluster dispose alors d'une nouvelle valeur de quota nouveau_quota si l'opération est autorisée, comme nous le verrons à propos de la protection des objets de Parx.

Chaque cluster est configuré suivant une topologie paramétrable. En fonction du support matériel disponible, nous avons vu que cette topologie pouvait être un graphe complet ou un graphe particulier (arbre, grille, hypercube ou réseau spécifique). Cette configuration peut être définie ou modifiée par :

```
t_error cluster_set_configuration ( t_cluster id_cluster,
                                    t_configuration configuration
                                  )
```

Une nouvelle configuration peut être soit une topologie quelconque, soit un graphe étiquetté par les numéros logiques des processeurs.

Les machines ne disposent pas toujours de dispositifs de configuration et de reconfiguration. Dans ce cas, seul un placement pourra être effectué une fois pour toutes à la création du cluster. Les primitives **cluster_add** et **cluster_set_configuration** doivent alors être unifiées. Il n'y a plus alors de raison de manipuler des processeurs sans tenir compte en même temps de leur configuration. Le type "t_processor_desc" doit être remplacé par une description globale de processeurs, chacun ayant des caractéristiques propres, connectés dans un réseau. "t_configuration" est enrichi pour refléter ces nouvelles informations.

La configuration courante d'un cluster peut être obtenue par la primitive :

```
t_error cluster_get_configuration ( t_cluster id_cluster, t_configuration
                                    configuration
                                  )
```

6.2 Les processus de Parx

Les systèmes d'exploitation traditionnels ne proposent en général qu'une seule abstraction pour l'exécution de programmes : le processus. Il regroupe à la fois les fonctions d'entité administrative, d'espace d'adressage, d'unité de parallélisme ou de pseudo-parallélisme, d'ordonnancement et enfin de flot de contrôle d'instructions. Les motivations de ce cadre unique pour autant de fonctions ont été abordées dans la section 2.2, et nous n'y reviendrons pas. Cependant, l'utilisation du parallélisme demande que cette assignation unique soit découpée, afin de pouvoir exécuter plusieurs flots d'instructions dans les différents domaines de chacune des fonctionalités.

Nous parlons de **parallélisme** et **pseudo-parallélisme** pour distinguer l'exécution simultanée sur des processeurs différents de sa simulation logicielle opérée par un noyau de système.

Nous parlons de **concurrence** et **collaboration** dans le contexte des politiques d'ordonnancement, pour distinguer des *groupes* de processus dont un seul peut être activé sans contrainte, de *familles* pour lesquelles rien ne sert d'exécuter l'un des membres si un autre est bloqué dans une opération [MaPu88].

6.2.1 Trois niveaux de processus

Une application parallèle, comme la plupart des programmes complexes nécessitant de fortes puissances de calcul, est un ensemble cohérent avec des besoins spécifiques, et doit être considérée comme tel. De nombreux projets, constatant la nécessité de cette cohérence, se voient contraints d'introduire des primitives permettant de manipuler des collections d'entités ou d'agir sur des données distribuées formant un tout logique dont la sémantique est hors de portée du système.

6.2.1.1 Tâche parallèle, tâche et thread

Parx intègre dans ses fondements mêmes cette cohérence en proposant une entité explicite, la **tâche parallèle** ou "Ptâche", permettant la manipulation du parallélisme et de la distribution. Cette approche semble de prime abord bien lourde pour les programmes de très courte durée de vie, mais nous verrons qu'ils sont supportés efficacement dans un environnement d'exécution adapté (voir section 6.2.3). Il correspondent à un grain d'exécution fin, la tâche parallèle étant le grain le plus gros.

Une tâche parallèle est un programme parallèle en cours d'exécution sur la machine abstraite correspondant aux besoins du programme : le cluster. Elle met en œuvre le modèle de programmation choisi par le programmeur. Elle est composée de **tâches** s'exécutant en parallèle sur des processeurs virtuels.

Un programme parallèle peut exprimer un degré de parallélisme quelconque, le système d'exploitation et l'environnement d'exécution se chargeant ensuite de lui allouer un degré effectif de parallélisme. Cette allocation se fait en fonction de deux critères. Le premier est le besoin réel de l'application en termes de puissance de calcul, besoin exprimé directement au niveau du langage ou après analyse par un algorithme d'allocation prenant en compte les charges de traitement et de communication [MEMT88]. Le second est le besoin réel en termes de puissance de communication, exprimé de manière analogue. Les tâches collaborent à la réalisation des différentes parties d'un problème.

Chaque tâche est à son tour un espace d'adressage dans lequel peuvent s'exécuter en pseudo-parallèle plusieurs flots de contrôle. Un tel flot est appelé **thread**, et correspond à un grain fin d'exécution.

Cette approche, illustrée par la figure 6.2, découple les fonctionnalités du processus traditionnel. La Ptâche est une unité administrative. La tâche est à la fois un espace d'adressage et une unité de parallélisme et d'ordonnancement. Enfin le thread est un flot de contrôle d'instructions, une unité de pseudo-parallélisme et d'ordonnancement.

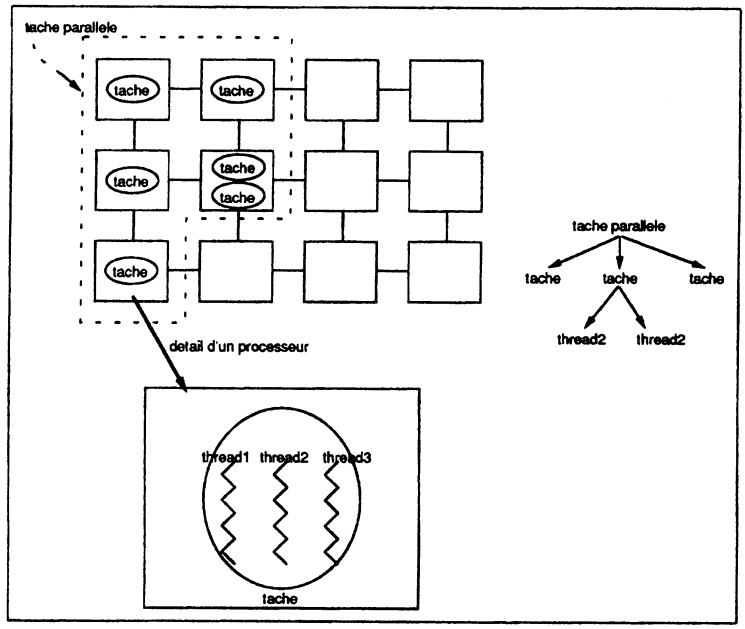


Figure 6.2: Ptâches, tâches et threads dans Parx

6.2.1.2 Rapport à d'autres systèmes

Situons brièvement ce modèle par rapport à ceux de Amoeba [Mullen90], Chorus [Chor87], Mach [TeRa87] et Unix.

Un processus Amoeba, un acteur Chorus ou une tâche Mach peut être représenté par une Ptâche comportant une seule tâche. Un thread Amoeba, Chorus ou Mach correspond à un thread Parx. Un processus Unix peut être une Ptâche comportant une seule tâche d'un seul thread.

Les facilités attachées à chacune de ces abstractions ne sont pas toujours directement disponibles sous Parx. La migration de processus Amoeba ou de ports Chorus, de même que les techniques sophistiquées de gestion de mémoire virtuelle de Mach, en sont des exemples. Il faut, dans ces cas, reconstruire les fonctionnalités ou, plus rarement, transformer les programmes application.

6.2.2 La tâche parallèle

La Ptâche regroupe un ensemble de tâches ne partageant pas de mémoire. Les tâches démarrent toutes "simultanément". La Ptâche termine soit par un appel explicite, soit lorsque toutes ses tâches ont terminé. De nouvelles tâches peuvent dynamiquement être créées dans une Ptâche.

Une tâche parallèle peut être suspendue et continuée, l'opération concernant chacune de ses tâches constituantes.

6.2.2.1 Opérations sur les tâches parallèles

Une Ptâche s'exécute sur un cluster et un seul, qui doit exister avant que la Ptâche soit créée. Plusieurs Ptâches peuvent s'exécuter sur le même cluster.

```
t_error Ptask_create ( t_cluster * id_cluster, t_Ptask * id_Ptask, t_boolean
                       persistance
                     )
```

Une tâche parallèle est créée sur le cluster id_cluster. L'identificateur de la tâche parallèle créée est retourné dans id_Ptask. Le but de cette primitive est de mettre en place les structures de contrôle nécessaires à l'administration et aux opérations sur la Ptâche créée. L'exécution du programme ne commence qu'avec la création de threads dans la tâche parallèle.

Lorsque le dernier thread d'un processeur alloué à une tâche parallèle termine, se pose le problème de l'utilisation du processeur désormais inoccupé. Une utilisation efficace des processeurs demande que le processeur soit remis à la disposition du système. Cependant, la Ptâche propriétaire peut, par la suite, décider de créer de nouvelles tâches sur ce processeur. Afin d'éviter le gaspillage par omission des ressources processeur, une Ptâche souhaitant conserver ses processeurs doit explicitement le demander. Le paramètre persistance indique si les processeurs doivent être libérés ou non.

```
t_error Ptask_delete ( t_Ptask id_Ptask
                     )
```

Cette primitive effectue la destruction d'une tâche parallèle précédemment créée. Les processeurs sont remis à la disposition des autres Ptâches du cluster.

```
t_error Ptask_suspend ( t_Ptask id_Ptask, int * nombre_tâches, t_task
                        tâche[]
                      )
```

permet de suspendre l'exécution d'une tâche parallèle. Toutes ses tâches sont suspendues. Le tableau tâche contient les identificateurs des tâches suspendues, et que l'on pourra continuer plus tard.

```
t_error Ptask_resume ( t_Ptask id_Ptask
                     )
```

permet de continuer une tâche parallèle précédemment suspendue.

Diverses fonctions utilitaires permettent d'acquérir de l'information sur les tâches parallèles.

```
t_error Ptask_info ( t_Ptask id_Ptâche, t_Ptask_info * information
                   )
```

permet de s'informer de l'état des tâches d'une tâche parallèle.

```
t_error Ptask_identity ( t_Ptask id_Ptask
                       )
```

permet à une tâche parallèle de connaître sa propre identité.

6.2.2.2 Graphe d'exécution d'une tâche parallèle

Une tâche parallèle est un programme parallèle en cours d'exécution, se décomposant en deux niveaux de parallélisme et pseudo-parallélisme. Le premier niveau est sa décomposition en un ensemble de tâches, chacune étant à son tour constituée de threads qui sont le second niveau. Les tâches constituent des processeurs virtuels et sont les sommets d'un graphe que nous appelerons "graphe d'exécution" de la tâche parallèle. Les arêtes du graphe représentent les communications entre tâches. Le graphe d'exécution d'une tâche parallèle varie avec la création et la terminaison de tâches.

Une tâche parallèle s'exécute sur plusieurs processeurs. Un certain nombre d'informations doivent diriger le placement de ses tâches constituantes sur un cluster de processeurs. Nous n'adressons pas ici le problème complexe de l'allocation de processeurs aux programmes, qui fait l'objet d'une thèse séparée.

Nous décrivons succintement quelques-unes des informations de placement que doit contenir un programme objet chargeable représentant une tâche parallèle. Des descriptions plus complètes se poursuivent dans le cadre de notre équipe de recherche, et des résultats ont été rapportés dans [Eudes90,Menn88,SnI89]. Une thèse est en cours sur le sujet.

La reconfiguration

Nous avons étudié l'usage de la reconfiguration dynamique dans la section 4.2.2. La présentation des paradigmes de programmation parallèle, notamment ceux de Carriero et Gelernter (section 2.1), justifie aussi l'usage de la reconfiguration. Celle-ci n'est pas toujours physique : l'exécution d'une tâche parallèle définit un graphe d'exécution dont la configuration logique peut être modifiée ; on parle alors de reconfiguration logique. Toutes les machines ne disposent pas de possibilités de reconfiguration physique, mais des mécanismes matériels et logiciels permettent d'en réaliser les fonctionnalités. Celles-ci sont essentiellement :

1. permettre l'expression parallèle de modèles de programmation demandant des topologies physiques adaptées à une application, ces topologies pouvant changer dans le cours de son exécution.

   Les paradigmes des parallélismes "agenda" ou "processor farm" rendent bien compte de ce besoin. Des données sont distribuées à un ensemble de processeurs, opération qui nécessite une certaine topologie. Ensuite, pendant le traitement, une autre topologie peut être nécessaire. D'autre part, dans le traitement agenda qui comporte plusieurs phases, chacune peut réclamer une topologie adaptée.

2. réduire les distances entre processeurs, ce qui revient généralement à diminuer le diamètre du graphe d'exécution, afin de réduire les temps de communication.

Les systèmes, distribués entre autres, sont confrontés depuis longtemps aux problèmes de configuration, résolus par la migration de processus et le partage d'objets en mémoire virtuelle.

Nous définissons trois modes de configuration. Nous parlerons de configuration **statique** lorsqu'une tâche parallèle sera exécutée avec le même réseau maillé de connexions entre ses tâches. Dès la création de la tâche parallèle, un tel réseau maillé est établi pour toute la durée de son exécution.

Nous parlerons de reconfiguration **quasi-statique** ou **synchrone** lorsque l'exécution d'une même tâche parallèle demandera successivement l'établissement de différentes configurations de réseau maillé. La succession des configurations peut être connue à l'avance ou bien apparaître au cours de l'exécution de la tâche parallèle. Chaque reconfiguration requiert une synchronisation de tout ou partie de l'ensemble des différentes tâches de la configuration précédente.

Nous parlerons de reconfiguration **dynamique** ou **asynchrone** lorsque l'exécution d'une même tâche parallèle demandera des changements locaux à sa configuration. Ces demandes sont imprévisibles et complètement asynchrones par rapport au déroulement global de la tâche parallèle.

Entre deux points de reconfiguration, l'échange d'informations entre processeurs non voisins nécessite l'acheminement de messages. Une reconfiguration peut avoir des répercussions plus ou moins importantes sur les informations de routage. Ces répercussions peuvent être très locales ou bien affecter plusieurs processeurs. Il faut dans le second cas synchroniser les processeurs ainsi affectés pour éventuellement modifier leur information de routage. On retombe alors dans une reconfiguration plus synchronisée. On peut partitionner les dispositifs de communication de chaque processeur, en réservant une partie à la reconfiguration asynchrone, l'autre au routage. Ceci a été réalisé à l'occasion d'une démonstration, dans le cadre du projet Esprit Supernode II, utilisant le système expérimental Mimics [Botteri91].

Le chargement

Un programme objet chargeable est constitué, du point de vue du chargeur, d'un ensemble de modules parallèles $M_1, \ldots M_n$ et d'informations de configuration. Les M_j sont des programmes pseudo-parallèles. Chaque module s'exécute sous la forme d'une tâche. Pour chaque module est fourni un ensemble d'objets à reloger avant exécution, entre autres des objets de communication, typiquement des

canaux. La liaison entre objets de communication ne pourra être achevée qu'après la décision du placement des modules parallèles sur les processeurs physiques.

Plusieurs modules peuvent être regroupés sur le même processeur. Le regroupement est effectué par un algorithme de placement en fonction d'une information de configuration. Son obtention a été expliquée en 6.2.1.1. Elle est constituée de deux parties. La première établit les liaisons formelles entre les objets de communication des modules M_j. Ils doivent tous être appariés. Une séquence d'appariemments peut être fournie, chacun correspondant à une étape de configuration. Dans ce cas, certains objets peuvent ne pas être appariés au cours de certaines étapes. La seconde partie de l'information sert à mesurer les poids respectifs des communications et à aider le système à former la topologie souhaitée, en projetant le graphe des tâches sur celui des processeurs physiques [Menn88].

La protection

La communication au sein d'une même tâche parallèle est complètement autorisée. Par contre les relations entre tâches parallèles de clusters différents sont contrôlées. Ainsi, toute communication entre tâches parallèles devra passer par l'intermédiaire du système. Pour détecter des communications illégales, il faut pouvoir isoler les tâches parallèles.

Dans le cas d'une machine physiquement reconfigurable, ceci peut être obtenu par la reconfiguration du cluster supportant la tâche parallèle. Il suffit par exemple de ne relier les liens de communication du cluster à aucun autre processeur. Lorsque ceci n'est pas possible, on peut établir une "ceinture sanitaire" mise en œuvre par des processeurs système. Il est cependant bien connu que la mise en place de ces barrières étanches est difficile, et souvent impossible [Tanen87].

Lorsqu'une tâche parallèle veut communiquer avec son environnement, il lui faut passer par l'intermédiaire d'un serveur de localisation. Ce dernier connaît un ensemble de services ainsi que leurs moyens d'accès au public. Elle peut ainsi entrer en relation avec d'autres tâches parallèles dites "serveurs".

6.2.3 La tâche

Elle représente un processeur virtuel et un espace d'adressage protégé. Une tâche ne peut dépasser les limites de la mémoire d'un processeur physique ; chacune est ainsi circonscrite à un seul processeur. Les différentes tâches d'une Ptâche contribuent au déroulement d'un programme parallèle. Elles sont actives simultanément, et forment ainsi des familles d'ordonnancement.

Les tâches de Ptâches partageant un processeur physique en ont chacune un pourcentage d'utilisation propre. Ce pourcentage de temps processeur est partagé entre les threads de chacune.

Des ensembles de tâches peuvent être créés, suspendus, continués et détruits. Ces opérations s'appliquent aux flots de contrôle s'exécutant dans leur espace d'adressage. La sémantique exacte des fonctions de suspension est plus ou moins lâche suivant l'état des flots de contrôle d'une tâche, qui peuvent être bloqués dans des appels systèmes.

De nouvelles techniques adaptées aux systèmes parallèles, notamment pour la création, l'activation et la terminaison de tâches, sont présentées.

Création et activation

A la création d'une tâche parallèle, plusieurs tâches sont créées et activées. Dans chaque tâche, un seul thread est activé. Nous définirons le thread à la section suivante.

La création d'une tâche signifie l'apparition d'une charge supplémentaire dans le système. Afin de répartir judicieusement cette charge, un algorithme de répartition statique de la charge peut alors être mis en œuvre. Les problèmes de placement et de répartition de charge sont étudiés dans notre équipe, et nous renvoyons à [TaMun90,TaMun91].

L'image mémoire d'une tâche peut être obtenue de plusieurs manières :

1. le code ou les données sont déjà présents sur le processeur (résidu d'une exécution précédente par exemple),
2. ils doivent être récupérés à partir d'une autre tâche déjà chargée. Ce schéma est utilisé par Unix.
3. ils sont présents sur un support de mémoire secondaire (typiquement dans des fichiers).

Le premier cas permet de réutiliser le code chargé sur un processeur pour une tâche donnée. On peut ainsi, pour des commandes exécutées fréquemment (compilation par exemple), conserver leur code en mémoire et se contenter de recharger les zones de mémoire correspondant aux données initialisées.

Pour ce faire, il suffit de connaître les zones de mémoire correspondant aux données initialisées. Lors d'une seconde exécution de ces tâches, que nous qualifions de "collantes", il suffira de recharger les données initialisées et de passer les nouveaux paramètres d'appel. Ces tâches "collantes" peuvent être la source de gains en performance considérables car elles réduisent le temps de chargement d'une tâche à celui de ses données initialisées. Leur inconvénient est l'occupation permanente d'espace mémoire.

A titre d'exemple, un interpréteur de commandes peut précharger le code des commandes le plus souvent utilisées. Ainsi, les commandes interactives sont supportées en diminuant la pénalité due à l'éloignement éventuel d'un support de mémoire secondaire.

Le fait de maintenir des copies de programmes en mémoire pose le problème de leur cohérence avec l'original du programme objet sur support secondaire. Diverses techniques sont utilisables pour propager les modifications du programme objet. Le serveur responsable du support secondaire peut générer des messages à destination de ses clients lors de la modification de programmes objets. Une autre solution consiste à laisser le soin au client de demander le rafraîchissement du code objet.

Nous n'élaborons pas plus sur ce sujet, auquel sont applicables les techniques de pagination et de va-et-vient classiques [ARSha89].

Le second cas permet de créer plusieurs tâches exécutant des portions de code identiques. Il permet d'implanter la migration par création de nouvelles tâches reprenant l'image mémoire des anciennes. Un mécanisme similaire au "fork" Unix peut être implanté ainsi.

Les deux premiers cas permettent de créer des tâches sans faire appel à un système de fichiers gérant un support de mémoire secondaire.

Le dernier cas se rapproche des mécanismes classiques. Il faut demander l'image mémoire à un serveur de fichiers.

L'exploitation de la souplesse de ces schémas demande la séparation entre les opérations de chargement et d'activation de tâches.

Terminaison

La terminaison d'une tâche peut être une action explicite reflétant l'accomplissement d'un algorithme, ou bien résulter d'événements inattendus.

Une terminaison anormale doit pouvoir être détectée : il s'agit soit d'une panne matérielle, soit d'un problème logiciel. Une approche préventive nécessiterait la preuve de la correction de chaque programme, ce qui est impossible en pratique ; il faut donc procéder par détection.

Deux cas peuvent se présenter. Dans le premier, la tâche effectue des actions observables par le système d'exploitation. Si celles-ci sont illégales et non récupérables, la tâche fautive est terminée. Dans le second cas, la tâche n'effectue aucune action observable. Son comportement est alors assimilé à un blocage (absence de progression) si l'état inobservable persiste pendant une durée correspondant à un délai de garde. A chaque tâche est associée une durée maximale

d'exécution. Lorsque celle-ci est dépassée, la tâche est forcée de terminer.

Lorsqu'une panne est détectée, il faut éviter qu'elle se propage au reste des applications. Il faut donc signaler cet événement aux programmes utilisateurs et aux serveurs mis en cause. Nous verrons dans les paragraphes suivants le traitement de ces événements. Il faut que les serveurs prévoient le traitement d'événements tels que la terminaison d'un de leurs clients ; de même que les clients doivent résister à la terminaison d'un serveur.

Une fois que l'information de terminaison est remontée, soit à un niveau où elle est traitée par un utilisateur, soit au niveau du système d'exploitation, il faut disposer d'outils pour effectuer son traitement. Le mécanisme de remontée est celui des exceptions. On peut y associer des procédures de traitement, chargées d'exécuter les opérations de libération de ressources consécutives à une terminaison. Une terminaison peut être remontée à d'autres tâches parallèles. En particulier, la terminaison de serveurs doit être signalée au serveur de localisation.

Une fois les tâches interlocutrices prévenues, il faut s'occuper des ressources allouées par le système d'exploitation et l'environnement *run time*. Il s'agit typiquement de segments de mémoire et d'informations d'administration.

La terminaison illustre de façon évidente la nécessité d'un protocole de diffusion entre tâches parallèles.

6.2.3.1 Opérations sur les tâches

Nous avons vu que les tâches étaient un cadre pour l'exécution de threads. La création de tâches met en œuvre le chargement de code et de données sur les processeurs.

Nous mentionnons ici l'existence de "*segments*", qui sont l'unité de gestion de la mémoire. Les segments peuvent contenir de l'information acquise après communication avec un seveur, par exemple de programmes objets, ou copiés à partir d'autres segments. Nous ne décrirons pas la gestion de la mémoire, qui est assez classique et en dehors de notre propos.

```
t_error task_create ( t_Ptask id_Ptask, int nombre_tâches, t_task_desc_list
                      liste_descriptions, t_task_list liste_tâches, t_boolean
                      persistance
                    )
```

permet de créer un ensemble de tâches sur une tâche parallèle id_Ptask. liste_descriptions décrit les zones mémoires, ou segments, de chacune des nombre_tâches tâches. Les segments ainsi décrits sont attribués aux tâches créées, et dont l'identification est retournée dans le tableau liste_tâches. Lors de la terminaison d'une tâche, les segments qui lui étaient attribués peuvent être libérés, c'est-à-dire que les zones

mémoires correspondantes perdent leur contenu. Ces zones mémoire peuvent alors être réutilisées comme support pour d'autres segments. D'autre part, il peut être utile de conserver l'information en mémoire, afin de pouvoir la réutiliser. C'est le cas des segments de code, pour la mise en œuvre des tâches collantes présentées dans les paragraphes précédents. Le paramètre persistance permet de spécifier ce comportement. Cette facilité peut aussi servir au déboggage "post-mortem" de tâches.

```
t_error task_delete ( t_Ptask id_Ptask, t_task id_task
                    )
```

permet de détruire une tâche précédemment créée. Suivant la persistance associée aux segments de la tâche, ces derniers sont supprimés et les zones mémoire correspondantes rendues disponibles.

```
t_error task_suspend ( t_Ptask id_Ptask, t_task id_task, int
                       number_threads, t_thread id_thread[]
                     )
```

permet de suspendre une tâche id_task. Les threads ainsi suspendus sont reportés dans le tableau id_thread. Certains threads dans un état particulier ne peuvent pas être suspendus immédiatement. Il s'agit notamment de ceux engagés dans des appels système, et qui ne sont donc pas observables ou interruptibles. Dans ce cas, la suspension sera différée jusqu'au prochain point interruptible. La tâche, ou seulement un sous-ensemble de ses threads, peut être continuée plus tard.

```
t_error task_resume ( t_Ptask id_Ptask, t_task id_task
                    )
```

permet de reprendre l'exécution d'une tâche précédemment suspendue. Il est possible d'obtenir des informations sur l'état d'une tâche par la primitive suivante :

```
t_error task_information ( t_Ptask id_Ptask, t_task id_task,
                           t_task_information * task_info
                         )
```

permet de connaître l'état d'une tâche et de ses threads. On peut ainsi par exemple savoir si une tâche ou ses threads ont étés suspendus.

Enfin, il est possible à une tâche de connaître sa propre identification :

```
t_error task_identity ( t_task * id_task
                      )
```

6.2.4 Le thread

Le système est construit autour d'une abstraction d'exécution représentant un flot de contrôle dans un programme : le processus léger, appelé thread. C'est l'unité élémentaire d'exécution, correspondant à un grain fin. Un thread se limite à un compteur ordinal et quel-

ques 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, de façon à réduire au strict nécessaire son temps de commutation de contexte.

De plus en plus de processeurs offrent un support matériel pour faciliter la commutation de contexte (Motorola 88000, i860, T800 [Inmos87a], T9000 [Pount90] etc.), et le thread peut être partiellement implanté directement dans le microcode du processeur, comme c'est le cas pour les transputers.

Un thread s'exécute dans le contexte d'une seule tâche, de sa création à sa terminaison. Un thread peut être créé, suspendu, continué et terminé. Dans une même tâche, ils sont en compétition pour l'usage du processeur.

Les threads d'une même tâche forment un groupe du point de vue de l'ordonnancement. Ils sont exécutés par tranches de temps, et soumis à préemption (figure 6.3).

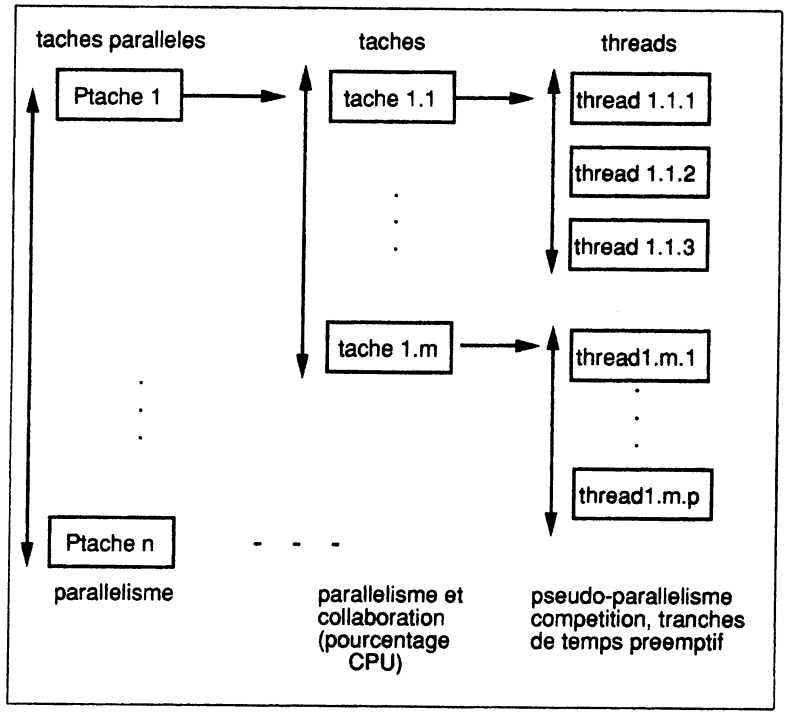


Figure 6.3: Ordonnancement de processus dans Parx

6.2.4.1 Opérations sur les threads

La création de threads effectue l'initialisation des registres du processeur, notamment le compteur ordinal, le pointeur de pile et les registres d'état. Les threads sont placés dans la file de l'ordonnanceur pour exécution.

```
t_error thread_create ( t_Ptask id_Ptask, t_task id_task, t_thread
                        id_thread[], int nombre_threads
                      )
```

permet de créer les nombre_threads threads décrits dans le tableau id_thread. Les threads sont créés dans la tâche parallèle id_Ptask dans l'espace d'adressage id_task.

```
t_error thread_delete ( t_Ptask id_Ptask, t_task id_task, t_thread id_thread
                      )
```

permet de détruire un thread.

```
t_error thread_suspend ( t_Ptask id_Ptask, t_task id_task, t_thread
                         id_thread[], int * nombre_threads
                       )
```

permet de suspendre un ensemble id_thread de threads. Leur description est mise à jour dans le tableau id_thread.

```
t_error thread_resume ( t_Ptask id_Ptask, t_task id_task, t_thread
                        id_thread[], int nombre_threads
                      )
```

permet de continuer des threads précédemment suspendus. Il n'y a pas de différence essentielle entre la création et la continuation, qui peuvent être réalisées par la même opération en fonction des facilités matérielles.

Enfin,

```
t_error thread_identity ( t_thread id_thread
                        )
```

permet à un thread de connaître sa propre identification.

6.3 Conclusion

Ce chapitre nous a permis de comprendre les motivations et les mécanismes de la construction de systèmes d'exploitation parallèles. Nous avons montré les limites d'approches à partir de systèmes existants.

Sur cette base, nous avons présenté un modèle de processus original et montré son utilisation pour des machines parallèles. Nous verrons dans la troisième partie que plusieurs systèmes d'exploitation pour le parallélisme adoptent une approche similaire, mais que Parx présente une cohérence tout à fait remarquable.

Le modèle de processus introduit dans ce chapitre, tout en s'inscrivant dans la lignée de la recherche actuelle, apporte un concept nouveau nécessaire à la programmation parallèle : la tâche parallèle. Les notions de tâche et de thread sont définies de façon cohérente à ce mode de programmation.

Un soin particulier a été accordé à la communication, qui recouvre la large majorité des protocoles usuels. Elle sera développée dans le chapitre suivant. L'ensemble permet de parcourir le spectre des

modèles de programmation présentés (voir section 2.1), et autour desquels se cristallise un large consensus. De même, les architectures présentées dans la section 3.1 peuvent supporter ce modèle.

Enfin, il n'est pas une simple agglomération de toutes les tendances existantes, et l'on observe de bonnes performances pour les prototypes en cours de développement et d'évaluation, comme nous le verrons au chapitre 8.

Chapitre 7

Le modèle de communication

A la suite de l'étude du chapitre 2, nous avons distingué deux grandes classes de communication : les données partagées et l'échange de messages. La première classe correspond à des machines à mémoire partagée. Certains projets [WSP90] implantent une mémoire virtuelle partagée sur des machines parallèles sans mémoire commune. Cette approche est valable pour un nombre de processeurs limité (quelques dizaines), et nous ne la suivrons pas. D'autres offrent différents niveaux de mémoire, l'une privée, l'autre accessible à travers l'un des réseaux étudiés en 3.2.2.1.

Il est intéressant d'envisager une gestion mémoire s'appuyant sur l'utilisation de zones mémoires non utilisées de processeurs voisins. Ces mécanismes s'apparentent à la régulation de charge, le taux de remplissage de la mémoire entrant en ligne de compte dans la mesure de la charge d'un processeur. Ils sont donc traités de concert avec la régulation de charge et font l'objet d'une thèse dans notre équipe de recherche. Ces aspects ne seront pas traités ici, même si nous mettons en place les mécanismes pour les supporter (voir section 6.2.3).

Notre modèle n'offre de possibilité de communication par données partagées qu'entre les threads d'une même tâche. Aucun support particulier, mis à part les mécanismes de synchronisation usuels sur les mono-processeurs, n'est proposé pour ce type de communication.

Nous concentrons notre propos sur l'échange de messages. Comme nous l'avons vu au chapitre 2, plusieurs protocoles de communication se rangent dans cette classe. Les conclusions de notre étude indiquent qu'il existe une famille de protocoles de base correspondant aux différents modèles, et qui permettent de bâtir simplement toutes les variations étudiées. Il faut bien comprendre que notre objectif premier n'est pas de réaliser des protocoles de communication. Il s'agit de mettre en place un noyau de communication permettant de supporter plusieurs protocoles différents simultanément. Pour illustrer cette capacité du noyau, nous avons choisi de réaliser un éventail de protocoles qui recouvre les besoins étudiés dans la première par-

tie. Les protocoles retenus couvrent un ensemble de récipients, de désignations d'émetteur et de récepteur, et de synchronisations.

Nous ne requérons comme support de communication dans la classe de machines visées que la communication entre processeurs connectés en réseau maillé. Nous l'avons vu dans la section 3.3.2 à propos du rapport entre les puissances de calcul et d'entrées-sorties : un processeur de communication qui puisse fonctionner indépendamment du processeur principal, pour la réception, le routage et l'émission, est indispensable pour les réseaux de taille importante.

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 correcte de construction de protocoles. Chaque protocole définit des objets de communication et un ensemble d'opérations.

Ces quatre protocoles sont supportés par un noyau de communication, qui réalise aussi la fonction de routage sans interblocage avec des caractéristiques originales d'efficacité et de prise en compte de différentes configurations. Ce travail, réalisé dans les équipes Sympa et Flop, a été publié dans [Langue89,MMS90,Gonza91]. Ce noyau constitue notre réalisation principale, et il intègre les fonctions de communication et la gestion de processus légers.

7.1 Le noyau de communication

7.1.1 Les problèmes de communication

7.1.1.1 Interblocage

Le premier et principal problème que rencontre le programmeur de machines parallèles est celui de la communication entre les différentes portions de son programme s'exécutant en parallèle. Les symptômes sont les suivants : le programme se comporte différemment lors d'exécutions successives, il "bloque" parfois sans raison apparente, les performances attendues ne sont pas au rendez-vous etc.

De multiples problèmes sont, à tort, collectivement regroupés sous l'appellation "interblocage" par les utilisateurs. Il y a évidemment l'interblocage lui-même, direct ou indirect, mais aussi la famine, les goulots d'étranglements semblables aux embouteillages automobiles, et l'accès inéquitable aux ressources, entre autres. Ces problèmes peuvent intervenir à n'importe quel niveau de la structure logicielle, et leur résolution dans une couche donnée ne préserve pas de leur réapparition dans une couche supérieure.

Le projet Parx a permis le développement d'un noyau de communication. Les problèmes de communication ont été clairement identifiés, et un ensemble de solutions étudiées et mises en œuvre. Différents algorithmes de routage sont proposés avec la même architecture du noyau. Chacun des modules du noyau résoud un ensemble de problèmes ; les interfaces sont bien définies et les contraintes de correction, ainsi que les paramètres de fonctionnement, établis.

7.1.1.2 Approche adoptée dans Parx

Le noyau de communication est, à la base, un ensemble de processus communiquants, collaborant à la réalisation de protocoles de communication de bout en bout dans une architecture à base d'échanges de messages. Il utilise des techniques largement répandues dans la littérature : contrôle de flux [GeKle80], canaux virtuels [DaSe87], combinées et modifiées dans un résultat unique original [Gonza91, MMS90]. L'objectif est de permettre la communication entre processus de la machine selon n'importe quel protocole, et ce quelque soit leur localisation physique.

A cet effet, il propose un ensemble de protocoles de base permettant de bâtir un riche ensemble de protocoles plus complexes. Les protocoles de base utilisent des mécanismes de routage qui constituent la première couche du noyau. Elle correspond au niveau réseau du modèle OSI, et effectue l'acheminement de messages de taille limitée, appelés *paquets*, correspondant à un dimensionnement des espaces tampons selon une estimation de la taille moyenne du message échangé. Chaque paquet est subdivisé en un *en-tête*, comportant des informations de routage et d'autres nécessaires à son protocole, et un *corps* qui constitue une partie du message transmis par l'utilisateur.

Comme la plupart des implantations de logiciels de communication pour machines parallèles, le noyau ne se conforme pas au modèle OSI, qui a été défini pour la communication sur réseaux locaux ou nationaux, et incorpore des fonctionnalités qui nous sont inutiles. Nous nous intéressons à des types de communication différents de par leurs caractéristiques de débit et de délai entre les communications.

7.1.2 Le routeur

7.1.2.1 Description

Le routeur, illustré en figure 7.1, est le programme acheminant les messages de proche en proche vers leur destination finale. Il utilise des chemins sans cycles [Gonza91,MMS90], de façon à ne pas générer d'interblocages. Cependant, pour assurer un service de routage correct, il doit respecter des contraintes additionnelles :

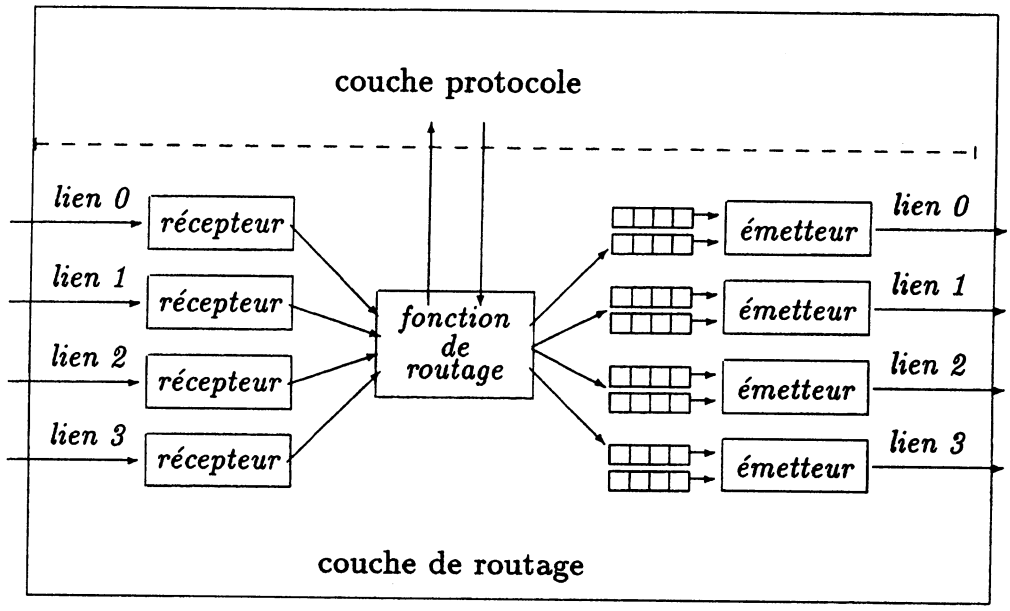


Figure 7.1: Structure du routeur

1. les messages émis doivent atteindre leur destination en un délai de temps fini. Ce délai est fonction de la longueur du chemin choisi et de la charge courante du réseau.

2. les messages ne sont jamais perdus. Une pratique courante dans les programmes de communication est de fournir un service de datagramme acheminant les messages, avec comme contrainte qu'ils soient délivrés au plus une fois [Mulder88]. L'usage de délais de garde et d'accusés de réception permet ensuite de construire des protocoles sans perte. Dans le cas des machines parallèles, les processeurs et media de communication sont "plus fiables" par hypothèse. D'autre part, l'usage de délais de garde entraîne la retransmission de messages et surcharge inutilement le réseau.

3. il n'y a pas de famine. Toutes les sources et destinations de messages bénéficient d'un service "équitable". Toute demande d'émission est traitée, sans attente indéfinie. Ce point est aussi une cause de bloquage.

4. la progression d'un message ne peut être conditionnée par l'acceptation d'un nouveau message par le routeur.

   Cette contrainte se reporte au niveau de l'interface du routeur, et est pénalisante comme nous le verrons dans la section 7.1.3. Ceci est une cause d'interblocage importante et pernicieuse. Le routeur fonctionne sans interblocage mais *son utilisation* sans précaution peut introduire des blocages. Nous expliquons dans la

section 7.1.3.2 le processus de formation de chaînes de dépendances aboutissant à un cycle représentant un interblocage.

Pour rester dans la généralité des machines parallèles, les paquets en provenance d'un même processeur ne sont pas forcément reçus dans l'ordre de leur émission. Nous pouvons ainsi supporter la modification dynamique des routes dans le réseau. Le routage adaptatif ou la reconfiguration sont ainsi pris en compte par le noyau. Les protocoles de communication réalisés admettent que les paquets s'entremêlent.

7.1.2.2 Schéma d'implantation

Le routeur a trois fonctionnalités principales, que nous décrivons brièvement.

Fonction de réception sur les liens

La première est celle de "**gardien de lien**". Les messages disponibles sur les liens d'entrée sont lus. Dans le cas du transputer, comme de beaucoup d'autres processeurs de machines parallèles reliés par des liaisons bi-point, il est indispensable d'effectuer cette lecture sous peine de bloquer le flux de messages entrants sur ce lien. Les liens du transputer fonctionnant en parallèle, il y a un gardien par lien d'entrée. Il fournit un tampon de réception pour le processeurs de communication qui fonctionne par accès direct à la mémoire, sans intervention du processeur principal. Ce mode d'opération, avantageux si le message est destiné au processeur récepteur, est pénalisant s'il doit être ré-acheminé. En effet, il interdit l'utilisation efficace des techniques de *Virtual Cut Through* [KeKle79] ou encore de *Wormhole* [DaSe87].

Le gardien **doit** délivrer chaque message reçu. Dans le cas où la délivrance ne pourrait se faire immédiatement, il dispose d'une fenêtre lui permettant de débloquer d'autres messages pour lesquels le prochain intermédiaire est disponible. Le nombre maximal de buffers utile pour cette fenêtre est celui des destinataires suivants possibles : le nombre de liens en sortie (moins un, un message ne ressortant normallement jamais par son lien d'entrée), plus le nombre de protocoles.

Fonction de détermination du lien de sortie

Une seconde est celle de "**routeur**". Il analyse l'en-tête de chaque message. En fonction de son champ destination, il effectue l'une des actions suivantes :

- si le message est arrivé à destination, il est transmis aux protocoles. Une condition nécessaire pour que le routage soit non

bloquant 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 autre condition à durée de réalisation indéfinie. Elle implique aussi que le contrôle de flux n'est pas réalisé dans le routeur, mais au niveau des protocoles.
Notre premier objectif étant l'efficacité, et dans l'absence de support de mémoire secondaire qui autoriserait de larges espaces de tampons, notre choix est justifié par le type d'applications que nous visons. Par hypothèse, la communication est intensive, et nous choisissons de n'effectuer le contrôle de flux strictement, que lorsque le protocole le juge utile, ce qui n'est pas généralement le cas. Nous améliorons ainsi très largement l'utilisation de la bande passante en communication. Lorsqu'un message est accepté par un protocole, il sort du réseau de routage.

- si le message doit être acheminé vers un autre processeur, le routeur choisit un lien de sortie approprié et passe le message à l'émetteur de lien correspondant.

Fonction d'émission sur les liens

La dernière fonctionnalité est celle d'**émetteur de lien**. Cette fonctionnalité est séparée du routeur pour lui éviter de se préoccuper des problèmes de la transmission, qui peut être bloquante. Il est essentiel de ne pas s'engager inconditionnellement dans l'émission d'un message, car cela est une condition de bloquage rapide de tout le réseau. Chaque lien de sortie dispose donc d'un émetteur de lien, et ils fonctionnent tous en parallèle suivant le même principe que les gardiens.

7.1.2.3 Evaluation

Il existe plusieurs variantes, dont nous avons expérimenté quelques-unes, à ce schéma de base. Par exemple, la fonction de routage peut être implantée directement dans les gardiens de liens. En fonction du processeur de communication disponible, ce schéma peut radicalement changer, les fonctionnalités demeurant cependant les mêmes. En particulier, l'arrivée de la famille de processeurs T9000 permettra de reporter la fonction de routage au niveau matériel. Il restera néanmoins à gérer dynamiquement l'utilisation efficace des chemins de communication.

Il apparaît clairement que le fonctionnement correct de l'acheminement de messages est conditionné d'une part, par la possibilité d'émettre sur les liens, et d'autre part de pouvoir délivrer les messages aux protocoles au bout d'un temps fini. En fait, ces deux conditions se ramènent à une seule, puisque l'émission sur un lien

est conditionnée par la réception du côté du processeur connecté, elle même dépendante soit d'une émission, soit d'un protocole. Il suffit donc de s'assurer que la panne d'un protocole est surmontable. Un délai de garde est associé à chaque gardien pour l'opération de passage d'un message aux protocoles. Ce délai expiré, le protocole est considéré comme défaillant. Notons que ce cas ne devrait pas se produire, les protocoles implantés étant corrects (!). Les causes possibles de mauvais comportement sont donc limitées à un programme utilisateur sortant de son espace d'adressage, mais ceci est un problème de protection.

Nous avons ignoré les problèmes que peut poser la panne d'un lien de communication. Comme nous l'avons déjà expliqué plus haut, cette hypothèse est improbable, mais un délai de garde peut aussi être associé à chaque émetteur de lien.

Nous ne présentons pas les mécanismes de routage, qui font l'objet d'une thèse séparée [Gonza91], jumelle de ce rapport.

7.1.3 Les protocoles

Les conditions imposées par l'acheminement de messages impliquent un cadre général d'écriture pour les protocoles. La plus contraignante est l'acceptation inconditionnelle de tout message en provenance du routeur. Le processus de réception doit donc être sans mémoire, avec une boucle principale d'acceptation de messages. Si le protocole dispose des ressources nécessaires pour traiter la requête, typiquement un correspondant local est prêt à lire le message, la condition de non bloquage est simple à réaliser. Cela n'est cependant pas toujours le cas. Lorsque le correspondant n'est pas prêt ou que le protocole admet plusieurs correspondants, se pose le problème de la mémorisation de la réception du message.

Diverses méthodes sont utilisées au gré des protocoles. Les paragraphes suivants présentent le cadre général d'implantation du noyau.

7.1.3.1 Comportement générique

Le comportement de chaque protocole, illustré à la figure 7.2, est encapsulé dans un ensemble de trois processus. Le **récepteur** reçoit les messages du réseau (en l'occurrence du routeur), le **serveur** interface le noyau avec des bibliothèques utilisateur, et l'**émetteur** transmet les messages sur le réseau (en l'occurrence au routeur). 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 programmes des utilisateurs.

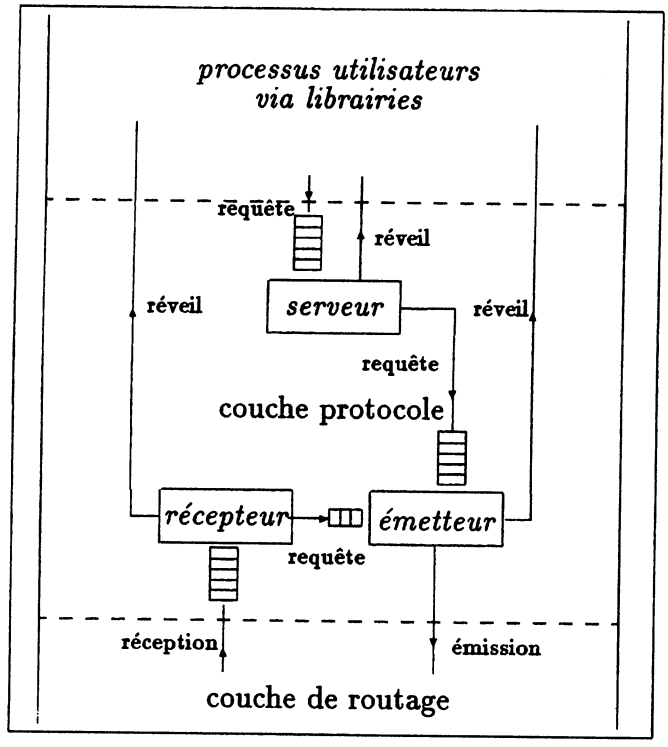


Figure 7.2: Structure de la couche protocole

Les files d'attente peuvent être bloquantes ou non bloquantes, chaque fois avec différentes politiques de retrait d'un élément :

1. premier entré, premier sorti. C'est la politique utilisée pour éviter la famine.
2. la file d'attente est multi-niveaux. Chaque niveau correspond à une classe de clients. Le premier client de chaque niveau est d'abord servi. C'est une variation de la politique précédente destinée à fournir un service équitable à chaque niveau.
3. la file est encore multi-niveaux, une priorité étant associée à chaque niveau. Un nombre λ (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. C'est une variation de la précédente introduisant des priorités.

Chaque protocole est décrit par un ensemble de couples (*numéro de requête, traitement associé*). Le numéro de requête, contenu dans l'en-tête du message, permet au protocole d'identifier et d'activer le traitement à lui faire subir. Les couples sont subdivisés en trois ensembles reconnus respectivement par le récepteur, le serveur et l'émetteur.

Cette implantation permet la modification dynamique des associations (numéro de requête, traitement). A la différence des "*streams*"

d'Unix [Bach86], qui permettent d'*empiler* et de *dépiler* des traitements à appliquer à un *flot de données*, notre implantation permet de *modifier* la réponse du protocole à un *type de requête* donné.

7.1.3.2 Contrôle de flux

Comportement typique

Nous avons vu en 3.2.2.1 que le projet iWarp [Bork89] distingue la communication par échanges de messages de la communication systolique. Le comportement généralement observé dans les systèmes parallèles est une combinaison des deux. Un chemin de communication est ouvert entre processus par l'attribution de descripteurs d'objets de communication (canaux, ports etc.). Les processus communiquent alors par échanges de messages pendant une durée arbitrairement longue avant de refermer le chemin. Par exemple, à la déclaration d'un canal occam, deux descripteurs sont alloués pour chacune des extrêmités en lecture et écriture. Pour une même communication, une fois que le rendez-vous est établi entre les correspondants, ils peuvent alors échanger des données sans qu'il soit nécessaire d'en contrôler le flux, puisque la réception est un puits.

Description du problème

Des interblocages peuvent apparaître au niveau des protocoles pour plusieurs raisons, dont en voici un exemple. Soit $G(S, A)$ un graphe décrivant un réseau de processeurs, avec $S = \{p, q\}$ et $A = \{e\}$. Supposons que p envoie un flot de messages à q, utilisant ainsi tous les tampons du chemin $(p, e) \longrightarrow (q, e)$. Simultanément, q envoie un flot de messages vers p, saturant tous les tampons du chemin $(q, e) \longrightarrow (p, e)$. Si le protocole sur p tient à envoyer le message suivant du flot *avant* d'accepter la réception d'un message quelconque de q, et que q fait de même, il y a interblocage au niveau des protocoles, même si l'acheminement de messages n'en a pas introduit.

Solution

Ce problème relève typiquement du contrôle de flux, et est du ressort des protocoles selon notre choix d'implantation. Il est résolu en évitant d'envoyer de flot de messages sans la garantie d'un puits du côté réception. Il faut aussi mémoriser la trace de messages n'ayant pu être traités immédiatement. Il suffit en fait de mémoriser leur émetteur et leur identification, le message entier étant conservé par l'émetteur tant que la requête correspondante n'est pas terminée. Ceci impose une borne supérieure au nombre d'émetteurs simultanés

vers un même récepteur. Ce problème ne se pose pas dans le cas des protocoles à un seul émetteur et un seul récepteur.

D'autres problèmes [GeKle80] sont la perte en efficacité consécutive à un gaspillage de ressources, et l'absence d'équité. Dans notre implantation, nous utilisons au mieux la bande passante des liens et minimisons le nombre de tampons utilisés. Les politiques de service des requêtes, exprimées en 7.1.3.1 sont équitables.

7.2 Les protocoles de communication

Nous détaillons les protocole d'accès mémoire, de rendez-vous et client-serveur qui ont été réalisés et démontrés.

7.2.1 Le protocole d'accès mémoire

7.2.1.1 Principe

Il est conçu pour réaliser les protocoles de bas niveau nécessaires au système d'exploitation et aux paradigmes d'appel de procédure et d'activation de processus à distance. Il permet à des threads de copier des portions de mémoire de n'importe quel processeur vers n'importe quel autre du réseau, s'ils en ont l'autorisation bien entendu.

Le récipient est une portion de la mémoire d'un processeur du réseau. Il contient un seul message, qui peut être de taille quelconque, et n'admet qu'un seul émetteur et un seul récepteur. La communication est initiée par un seul des protagonistes, qui fournit un descripteur et en identifie un autre sur l'autre processeur. La connaissance de descripteurs est dynamique, et l'on peut changer l'affectation d'un descripteur. Un ou deux descripteurs identifient une communication. L'un peut se trouver sur le site émetteur, l'autre étant alors sur le site récepteur et vice-versa (voir figures 7.2.1.1 et 7.4).

Le descripteur de communication, indique son état d'avancement. Le protocole est asynchrone et l'activation de threads sur les processeurs correspondants peut être associée à la terminaison de la communication.

7.2.1.2 Description

Le descripteur d'une communication comporte notamment des informations sur le nombre d'octets restant à transférer, et le thread à activer lors de la terminaison du transfert. Ce descripteur est consultable localement, par l'intermédiaire de fonctions appropriées.

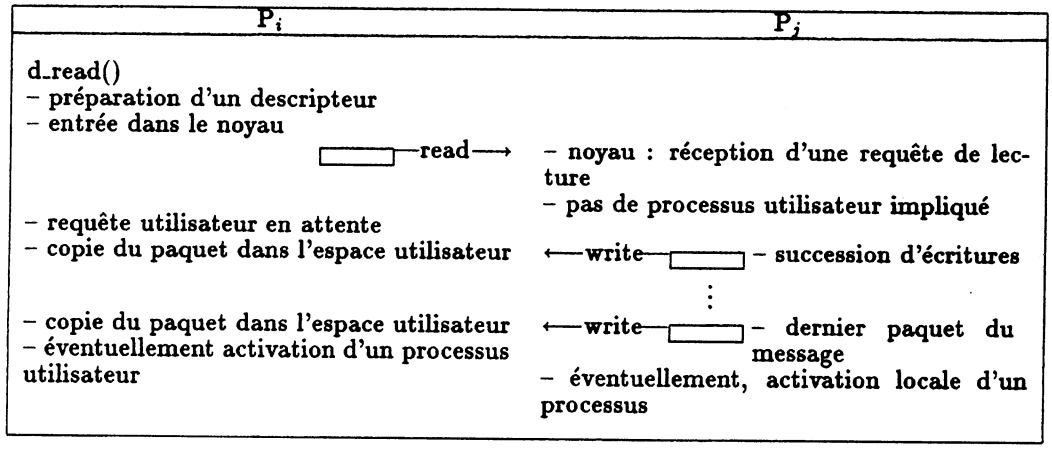


Figure 7.3: Exemple de déroulement de la primitive *read*

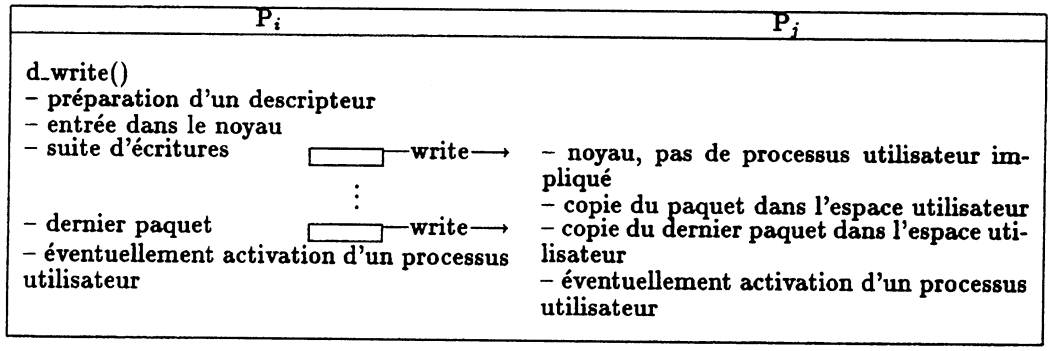


Figure 7.4: Exemple de déroulement de la primitive *write*

Chaque fonction est conçue comme une action principale, suivie d'un ensemble non exclusif d'actions complémentaires. Les actions principales sont l'écriture ou la lecture, les actions complémentaires étant la mise à jour du descripteur de communication, l'activation à distance d'un thread dont le descripteur est contenu soit dans l'en-tête du message, soit dans le descripteur local de communication. D'autres actions peuvent être facilement ajoutées.

Un premier ensemble de fonctions opère en un seul message du niveau routage. Il permet de lire ou d'écrire un message de la taille du tampon du routeur sur n'importe quel processeur sans qu'il soit besoin de la participation d'un autre programme du processeur cible.

Un second ensemble requiert que chaque processeur cible soit préparé à traiter une copie distante. Le demandeur désigne un descripteur de communication distant. Ce dernier est utilisé pour effectuer les actions locales demandées dans la requête. Il permet aussi de transférer des données d'un processeur à un autre à partir d'un troisième processeur.

Ce protocole se combine harmonieusement avec les autres. En effet,

il permet de transférer de larges portions de mémoire avec un coût minimum entre processeurs distants de façon asynchrone. Un serveur peut ainsi recevoir une requête mettant en jeu de gros volumes de données, la traiter et effectuer les copies de façon asynchrone et en parallèle, sans faire attendre inutilement le demandeur. Ce dernier a aussi la possibilité de s'enquérir de l'avancement du(des) transfert(s) et de se synchroniser sur sa(leur) terminaison à tout moment.

Ce protocole est utilisé dans le chargement dynamique de données ou de threads, et l'activation de ces dernières. Il a aussi des applications à la migration ou à l'implantation d'appel ou d'activation de procédures à distance (voir figure 7.5).

```
procedure migration()
{
.   thread = /* description d'un thread */
.   code = thread.debut_code;
.   taille_code = thread.taille_code;
.   d_write(processeur_cible, adresse_segment_code, code, taille_code);
.   donnees = thread.debut_pile;
.   taille_donnees = thread.taille_pile;
.   donnees = reloger(donnees, taille_donnees, adresse_segment_donnees);
.   d_write_and_activate(processeur_cible, adresse_segment_donnees,
donnees, taille_donnees);
}
```

Figure 7.5: Exemple de migration

7.2.1.3 Primitives d'accès

Deux primitives permettent de lire ou écrire dans la mémoire d'un processeur sans jamais nécessiter la contribution d'un processus utilisateur sur le processeur cible.

```
t_error d_peek ( t_Ptask id_Ptask, t_processor id_processor,
                 t_address adresse, t_byte buffer[], int buffer_size
               )
```

permet de lire buffer_size octets de l'adresse adresse du processeur identifié par id_processor dans la tâche parallèle id_Ptask. Le résultat, placé dans buffer, ne peut excéder la taille d'un paquet de la couche de routage, sous peine de nécessiter l'allocation d'un descripteur de connexion sur le processeur cible. Ceci entraînerait une gestion de mémoire dynamique, et que l'on ne saurait borner, dans le noyau de communication. D'autre part, d'autres primitives remplissent cette fonction. Les espaces de mémoire manipulés dans le noyau sont toujours alloués par un utilisateur.

```
t_error d_poke ( t_Ptask id_Ptask, t_processor id_processor,
                 t_address adresse, t_byte buffer[], int buffer_size
               )
```

permet d'écrire une information contenue dans buffer à l'adresse du processeur id_processeur de la tâche parallèle id_Ptask. A nouveau, la

taille de l'information, donnée par buffer_size, ne peut dépasser celle du paquet.

Les primitives suivantes permettent d'opérer sur des tailles de données illimitées. Elles nécessitent cependant la connaissance d'un descripteur de communication sur les processeurs émetteur et récepteur.

```
t_error d_read ( t_Ptask id_Ptask_source, id_Ptask_dest,
                 t_processor id_processor_source, id_processeur_dest,
                 t_descripteur descripteur_source, descripteur_dest
               )
```

permet de lire de l'information décrite par descripteur_source du processeur processeur_source de la tâche parallèle id_Ptask_source vers la destination (descripteur_dest, processeur_dest, id_Ptask_dest).

```
t_error d_write ( t_Ptask id_Ptask, t_processor id_processeur,
                  t_descriptor descripteur_source, descripteur_dest
                )
```

permet d'écrire de l'information du processeur appelant vers un processeur décrit par (descripteur_dest, processeur, id_Ptask).

7.2.2 Le protocole de rendez-vous synchrone

7.2.2.1 Description

Le protocole occam en est un représentant typique. Les objets de communication sont des canaux uni-directionnels point à point. Un canal permet à un couple de threads d'échanger des messages de taille quelconque, dans un seul sens et de façon synchrone. Il est illustré en figure 7.6.

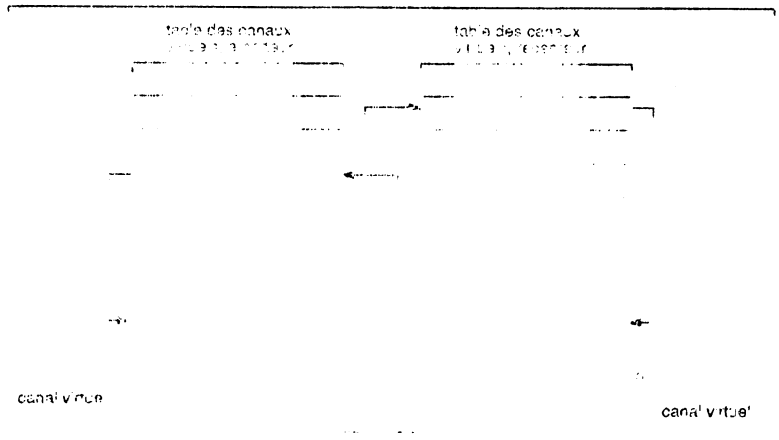
Les différents correspondants ayant des vitesses d'exécution relatives non déterminées, une construction de sélection non déterministe est offerte pour arbitrer la réception sur des canaux multiples.

Cette construction réalise l'alternative occam. Elle est implantée de façon *non déterministe*, mais beaucoup d'utilisateurs confondent non déterminisme et choix *équitable*, et ont souvent besoin de ce dernier quand ils utilisent l'alternative. Un support est aussi disponible pour implanter la sélection équitable (même probabilité lors de chaque tirage).

7.2.2.2 Algorithmes

Nous considérons un processeur et un processus émetteurs, et un processeur et un processus récepteurs. Le noyau exécute l'algorithme suivant pour le processus émetteur :

```
si le canal a une communication en cours (recepteur pret)
debut
  copier notre message vers le recepteur, a concurrence de la taille
```




```
  sinon fin emission
fin
sinon
  emettre une requete d'emission vers le recepteur, et mettre
  l'emetteur en attente de reponse;
fin
```

Chaque requête d'émission est accompagnée de données d'anticipation que l'on peut ajuster à la taille moyenne des messages transmis, de façon à effectuer le rendez-vous en deux messages seulement au lieu de trois au moins. L'inconvénient est que ces données d'anticipation doivent être conservées dans la mémoire du processeur destination.

Le noyau exécute l'algorithme suivant pour le processus récepteur :

```
si le canal n'a pas de communication en cours
  attendre l'initiative de l'emetteur : mettre le recepteur en attente;
sinon
debut
  lire le contenu du cache;
  si ce contenu satisfait a notre demande
  debut
    noter l'etat de l'emetteur (taille restante de son message);
    reveiller le processus recepteur;
  fin
  sinon
  debut
    (nous avons consomme la totalite du cache)
    envoyer un ACK a l'emetteur, lui notifiant la taille que nous
    reclamons;
  fin
```

```
fin
```

Les processus du noyau exécutent différents algorithmes lors de la réception de messages, en fonction du type de ces messages. Sur une réception d'ACK en provenance du récepteur :

```
si le message demande des donnees
debut
  envoyer les donnees de l'emetteur;
  si tout le message de l'emetteur est envoye
    reveiller le processur emetteur;
  sinon
    envoyer une nouvelle requete d'emission a la suite des
    donnees que le recepteur etait pret a recevoir;
fin
sinon
debut
  si l'emetteur n'a plus rien a envoyer
    le reveiller;
  sinon
    envoyer une nouvelle requete d'emission;
fin
```

Sur réception d'un message de requête d'émission provenant de l'émetteur, les processus du noyau exécutent l'algorithme suivant :

```
si le processus recepteur n'est pas encore au rendez-vous
debut
  memoriser la demande d'emission;
  copier le message d'anticipation dans le cache;
fin
sinon
debut
  en fonction de l'etat du processus recepteur :
  si debut d'alternative
    mettre l'etat du processus recepteur a fin d'alternative;
  si attente en alternative
    le reveiller;
  si fin d'alternative
    trop tard pour nous : signaler notre message qui sera pris
    en compte lors de la prochaine alternative;
  sinon
  debut
    copier le message chez le processus recepteur;
    si le recepteur se satisfait du message d'anticipation
      debut
        copier le restant du message d'anticipation dans le cache;
        le reveiller;
      fin
    sinon
      envoyer un message d'ACK en precisant la taille demandee;
  fin
fin
```

Sur réception d'un message de flot de données en provenance de l'émetteur, les processus du noyau exécutent l'algorithme suivant :

```
copier les donnees chez le processus recepteur;
si ces donnees sont les dernieres d'un message
    reveiller le processus recepteur;
```

Le protocole sommairement décrit présente deux particularités en dehors du rendez-vous classique. La première est qu'il permet d'échanger des messages de tailles variables. Le premier processus réveillé, que ce soit l'émetteur ou le récepteur, est mis au courant de la taille du message que son correspondant souhaite encore transmettre. Ce correspondant est bien entendu en attente jusqu'à ce que son message soit lu. Cette facilité est utile à la réalisation de processus serveurs, et permet par exemple de résoudre très élégamment le problème classique du multiplexage des liens des transputers. Il suffit que le processus multiplexeur lise *une partie seulement* du message de chacun de ses clients, qui peuvent alors utiliser des primitives de communication classiques, sans se préoccuper de la présence d'un multiplexeur.

La seconde est que l'adressage des correspondants se fait de façon transparente dans tout le réseau. Nous introduisons pour cela des *canaux virtuels*, qui sont une généralisation distribuée des canaux de communication. Un canal virtuel est un ensemble de deux extrêmités. Un couple (identificateur de processeur, numéro de canal) est une extrêmité de canal virtuel, et nous voyons que c'est ainsi que les canaux sont appariés par le noyau. Remarquons que c'est aussi la technique choisie par le processeur de communication intégré sur les processeurs T9000 d'INMOS. Sur chaque processeur, une table d'extrêmités de canaux virtuels permet de diriger les messages vers leur processeur destinataire. Cet appariement peut être établi et modifié dynamiquement.

7.2.2.3 Primitives d'accès

Un canal de communication possède deux extrêmités : l'une émettrice, l'autre réceptrice. Une description complète en est donnée dans [Gonza91]. L'initialisation des extrêmités requiert leur appariement, c'est-à-dire leur connaissance mutuelle par une opération de liaison. Celle-ci est donnée par un numéro de processeur et le numéro d'une extrêmité. Cet appariement peut être statiquement décidé au moment de l'édition de liens et du chargement, et il n'est donc pas prévu de serveur de localisation pour l'appariement dynamique de canaux. Par contre, il peut être changé afin de réutiliser une extrêmité pour désigner une autre communication. Ce mécanisme peut être utilisé comme base pour la migration.

```
t_error channel_setup ( t_channel canal, t_channel_type type,
                        t_processor proc_distant, t_channel canal_distant
                      )
```

permet d'initialiser ou de réinitialiser une extrêmité de canal. Le canal aura ensuite pour correspondant le canal de numéro canal_distant sur le processeur proc_distant, et pourra être utilisé pour les opérations compatibles avec le type donné.

```
t_error channel_in ( t_channel canal, t_byte buffer[], int taille_buffer
                   )
```

permet de lire des données sur le canal canal, dans le tableau buffer de taille taille_buffer.

```
t_error channel_out ( t_channel canal, t_byte buffer[], int taille_buffer
                    )
```

permet d'écrire des données sur le canal canal, à partir du tableau buffer de taille taille_buffer.

```
t_error channel_alt ( t_channel canaux[], int taille_canal, t_channel elu
                    )
```

permet de choisir un canal elu parmi les taille_canal du tableau canaux.

```
t_error channel_select ( t_channel canaux[], int taille_canal, t_channel elu
                       )
```

permet de choisir, de façon équitable, un canal elu parmi les taille_canal du tableau canaux.

```
t_error channel_status ( t_channel canal, t_chan_status etat
                       )
```

permet de connaître dans etat l'état d'un canal canal.

7.2.3 Le protocole client-serveur

7.2.3.1 Principe

Il est conçu pour la communication entre plusieurs clients et un serveur recevant leurs requêtes. Les objets de communication sont des ports uni-directionnels. Chaque port est une file d'attente de messages. Un port est localisé sur le site de la tâche réceptrice. Il débite son contenu 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 est bloquante si aucun récepteur n'est prêt. La réception est bloquante si aucun message n'est disponible (voir figure 7.8).

Les ports sont des objets dont la publication est assurée par un gestionnaire public de ports. Un serveur peut se déclarer disponible pour rendre un certain type de service, fournissant alors au gestionnaire public un ensemble de ports par lesquels il est accessible. Plusieurs serveurs peuvent rendre le même service.

Un client peut demander au gestionnaire de ports, universellement connu de même qu'un certain nombre de services de base, l'identification d'un port d'accès à un service donné. Moyennant les vérifications de droits d'accès et après le choix du serveur "le plus approprié" selon la requête, le gestionnaire fournit un port d'accès. Des critères tels que la proximité physique ou la charge des serveurs peuvent être utilisés pour satisfaire la requête [Shizgal87]. Ce port sera utilisé pour toutes les requêtes du client au serveur. Les ports peuvent aussi être

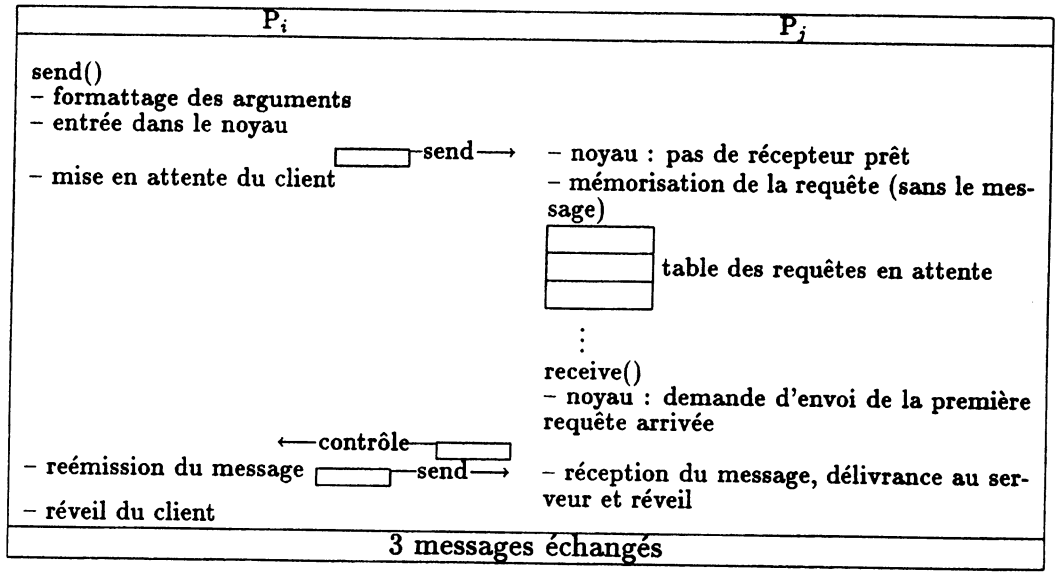


Figure 7.7: Exemple de déroulement de la primitive *send*

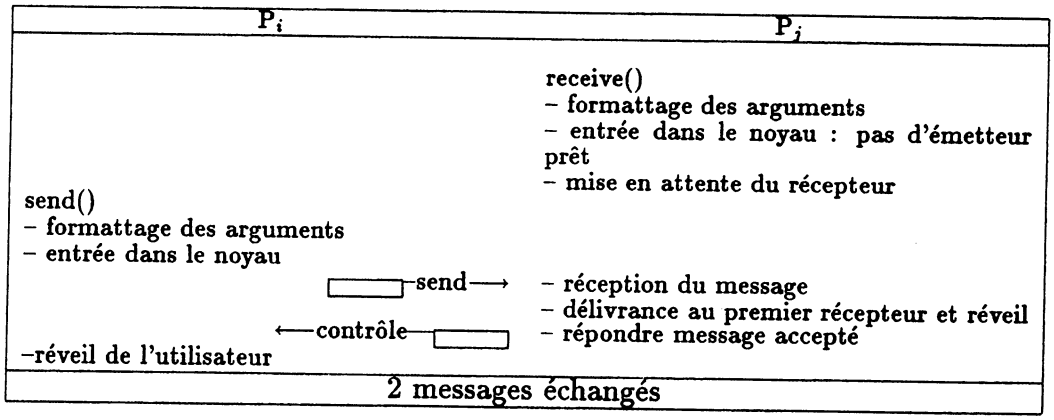


Figure 7.8: Autre exemple de déroulement de la primitive *send*

supprimés.

7.2.3.2 Description

Ce protocole est intéressant parce qu'il illustre les problèmes rencontrés lorsque les acteurs d'une communication ne sont plus réduits à un couple.

A la création d'un port, le créateur fournit le nombre maximal de requêtes d'écriture simultanées qu'il peut supporter. Les récepteurs et les émetteurs sont mémorisés. Lorsqu'une requête de réception est soumise,

- si la file d'attente des émetteurs n'est pas vide, la première requête est satisfaite. La requête est normalement mémorisée sur un pro-

cesseur, et il faut que son message soit retransmis. Au total trois messages auront été échangés dans ce cas.

- si la file d'attente est vide, la requête est mise en file d'attente en réception.

Lorsqu'une requête d'émission parvient à un port,

- si la file des récepteurs est vide, la requête est mise en file d'attente de réception.
- sinon, le message est immédiatement reçu par le premier des récepteurs. Au total deux messages seulement sont échangés.

Les serveurs ayant toute liberté de créer autant de threads qu'il en faut pour assurer leur service, le second cas est le cas général, aux contraintes d'espace mémoire près.

Alors que l'émetteur de protocole est bloqué dans une tentative pour soumettre une requête ou y répondre, plusieurs requêtes ou réponses peuvent parvenir au récepteur. Dans le pire des cas, il lui faut répondre à toutes. Pour répondre, il lui faut attendre que l'émetteur (en fait le lien de communication) soit libre. Cependant, il lui faut continuer à recevoir les messages du routeur (condition de non bloquage exprimée en 7.1.3.2). Si l'ensemble des émetteurs possibles est non borné, le récepteur peut être amené à réclamer un espace de mémoire temporaire non borné. Pour prévenir cette situation, nous plaçons une borne supérieure au nombre d'émetteurs utilisant simultanément le même port, et conservons les requêtes dans l'espace mémoire des émetteurs. Ainsi, il suffit au récepteur de conserver la trace de la réception d'un message, pour éventuellement en réclamer la répétition lorsque les conditions seront plus propices.

7.2.3.3 Primitives d'accès

Les ports, contrairement aux canaux, sont des objets dynamiques qui doivent être explicitement déclarés et supprimés. Ils sont déclarés par l'envoi d'un message au serveur de ports. De même, un processus utilisateur peut réclamer un port pour un service en envoyant un message au serveur de ports.

Un serveur peut déclarer un port par la primitive :

```
t_error port_publish ( t_service clef, t_port port[], int nombre_ports, int
                       max_clients
                     )
```

qui permet de rendre publics les nombre_ports ports contenus dans port. Le service correspondant est identifié par clef. Chaque port peut acceuillir au maximum max_clients simultanément.

Un serveur peut annuler la déclaration de ports par la primitive :

```
t_error port_unpublish ( t_service clef, t_port port[], int nombre_ports
```

```
                    )
```

qui permet de retirer du domaine public les nombre_ports ports contenus dans port. Le service correspondant est identifié par clef.

Un client peut accéder à un port par la primitive :

```
t_error port_get ( t_service clef, t_port * port
                 )
```

qui permet d'obtenir l'identificateur d'un port correspondant au service identifié par clef.

```
t_error port_unget ( t_port port
                   )
```

permet de fermer la connexion avec un port.

```
t_error port_send ( t_port port, t_byte message[], int taille_message,
                    int * taille, t_time timeout
                  )
```

permet d'envoyer un message message de taille taille_message vers un port. Un délai de garde timeout permet de borner l'attente à l'émission. taille permet de connaître la taille réellement échangée.

```
t_error port_receive ( t_port port, t_byte message[], int taille_message,
                       int * taille, t_time timeout
                     )
```

permet de recevoir un message sur un port. Un délai de garde permet de borner l'attente à la réception. taille est la taille effectivement lue.

```
t_error port_select ( t_port table_port[], int nombre_ports,
                      t_port * port_choisi
                    )
```

permet à un serveur de sélectionner l'un de ses ports disposant de messages dans le tableau table_port.

```
t_error port_status ( t_port port, t_port_status * status
                    )
```

permet à un serveur de connaître l'état de l'un de ses ports.

7.2.4 Le protocole de diffusion

Une modélisation en a été faite dans [Langue87]. Deux modes de diffusion sont étudiés : 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'il sont à l'écoute* de cette communication ; il y a synchronisation lâche. Il est possible de combiner les deux modes.

L'émetteur fournit un ensemble de récepteurs, qui est transformé en listes de diffusion. Chaque liste est le résultat de l'intersection d'un élément de la partition des processeurs formée en considérant l'ensemble atteignable par chaque lien physique, et de l'ensemble des récepteurs. Chaque liste est ainsi acheminée par vagues atteignant

progressivement toutes les parties du réseau. Ce protocole rappelle celui utilisé dans Linda et présenté dans la section 2.4. Cependant, celui de Linda utilise seulement deux parties.

Le récepteur indique sa participation à une diffusion en nommant l'émetteur.

Ce protocole n'a pas encore été implanté.

7.3 Conclusion

Ce chapitre nous a permis de présenter le modèle de communication de Parx. Celui-ci découle des études des chapitres 2 et 3, qui font apparaître la nécessité de la diversité de protocoles de communication. La gestion des objets de la mémoire, ainsi que le traitement des exceptions, bien qu'étudiées par ailleurs, n'ont pas été abordés, parce qu'ils n'étaient pas au cœur de nos recherches. La gestion des objets distribués nous paraît un domaine très prometteur pour les machines parallèles. En effet, elle est apparue nécessaire dans l'étude des modèles de programmation, et nous avons indiqué des pistes originales de recherche, notamment la collaboration de processeurs voisins.

Enfin, nous avons présenté le modèle de construction de noyaux de communication adopté dans Parx. Un aspect important est la généricité de la conception qui permet d'y insérer n'importe quel protocole. L'ajout ou la modification dynamiques de protocoles est aussi possible.

Le chapitre suivant présente une évaluation des performances du noyau de communication.

Chapitre 8

Evaluation des performances du noyau de communication

Nous donnons ici les résultats obtenus lors de mesures de performances du noyau de communication. L'étude de performances en environnement parallèle est un sujet nouveau, et les paramètres à mesurer ne sont pas encore clairement établis. C'est pourquoi nos mesures ont principalement visé à mettre en place un ensemble d'outils de mesure et à évaluer l'impact de différents paramètres du noyau de communication.

Les résultats que nous relatons ici le sont donc dans l'objectif unique de comprendre le fonctionnement du noyau. Nous n'avons pas développé notre prototype dans le but de produire un code exécutable optimal, mais plutôt de permettre un large éventail d'expérimentations qui mèneront à des noyaux optimisés pour des situations précises. Nous mettons donc en avant la méthode employée et la validation de nos instruments de mesure. Effectuer des optimisations sur le noyau sans connaître aussi précisément que possible son comportement réel nous paraît inutile. Nous insistons sur les paramètres nous permettant de comprendre d'éventuelles mauvaises performances, leur explication et, si possible, leur solution.

Quelques études de performances des logiciels de communication sont parues à ce jour. Elles nous paraîssent souvent incomplètes parce qu'elles avancent des chiffres sans toujours préciser les conditions de leurs mesures, ou se placent dans des conditions idéales qui ne correspondent pas à des situations réelles.

Nos mesures ont été effectuées sur des Tnodes de Telmat, comportant seize processeurs de travail transputer T800 et un contrôleur T414. Les liens de communication étaient configurés à 10 Mbits/s. Les programmes ont été compilés avec le compilateur C développé à l'IMAG. Ce dernier, destiné à des développements systèmes, conserve invariant le registre de pile. Ceci entraîne un supplément de deux cycles pour la plupart des références mémoire. Un cycle dure 50 ns, et dix accès

mémoire correspondent à un surcoût d'une microseconde. Ce choix a l'avantage de permettre une gestion plus simple des processus légers, mais il divise par un facteur de l'ordre de 2 à 3 la qualité des performances obtenues. Les temps que nous donnons sont donc à prendre sous cette contrainte.

Nous rapportons aussi quelques mesures effectuées avec le même environnement matériel mais en utilisant le compilateur C INMOS. Ces mesures montrent une très nette amélioration des performances, alors que le noyau n'a pas été modifié.

Nous détaillons 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.

8.1 Mesure d'un protocole

Nous avons choisi, dans un premier temps, de mesurer certains paramètres pour un protocole. Nous nous limitons à un protocole parce que nous ne prétendons pas effectuer une étude de performances complète, mais cherchons plutôt à comprendre le comportement du noyau de communication et l'influence de certains de ses paramètres. Nous avons choisi d'étudier le protocole d'accès mémoire à distance parce qu'il est le plus dépouillé de complexités liées à des spécificités de protocoles, et reflète bien la mécanique du noyau.

La primitive de base est `d_write`, qui permet d'échanger des données entre processeurs et de réveiller un processus distant sur la fin d'une transmission. Les temps mesurés commencent juste avant l'appel de la fonction de bibliothèque préparant les paramètres de l'appel au noyau, et terminent juste après que le message ait été entièrement reçu et un processus distant activé. C'est donc un délai d'utilisateur à utilisateur.

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. Deux liens de communication seulement ont été utilisés pour les expériences suivantes.

8.1.1 Influence des processeurs intermédiaires

La figure 8.1 montre le débit des messages, en fonction du nombre de processeurs intermédiaires, pour différentes tailles de messages. Des processeurs voisins ont un nombre nul d'intermédiaires. Aucun des processeurs n'exécutait de programme application lors de notre expérience. Nous voyons que le débit est une fonction croissante de

la taille des messages échangés. Le débit maximum constaté avec le routage en place correspond approximativement à celui atteint en utilisant directement les liens pour des messages de 16 octets.

Ces performances, tout en étant dans la moyenne des résultats obtenus dans les mêmes conditions par d'autres noyaux, ne sont pas très satisfaisantes. Rappelons que nous ne nous sommes pas intéressés aux performances dans l'implantation évaluée. Il faut tenir compte aussi des choix effectués par le compilateur, qui doublent pratiquement le temps de traitement. Lors d'une version précédente, écrite en assembleur pour des raisons de performances, nous avions obtenu les meilleurs résultats connus à ce jour, tout en préservant des propriétés importantes comme l'absence d'interblocage, une consommation en mémoire réduite et des routes proches des plus courts chemins. La version que nous évaluons, destinée à l'expérimentation, est écrite de manière très générique, et permet un comportement dynamique tout à fait intéressant (installation de protocoles, changement de fonctionnalités d'un protocole sans l'interrompre, etc.). Néanmoins, elle peut être améliorée, notamment par la mise en place d'un pool de tampons mémoire d'anticipation pour la réception de messages. Ainsi, un processeur doit recevoir des messages sur le même lien jusqu'à ce que *toutes* les sorties (liens d'émission et processus récepteurs des protocoles) soient indisponibles, ou que les tampons soient tous pleins. Ce n'est que dans ces conditions qu'il se bloque. Dans la version mesurée, il attend dès qu'une sortie est indisponible.

La figure 8.2 montre le débit en fonction du nombre de processeurs intermédiaires alors que chaque processeur subit une charge de 75%. Nous pouvons constater que le débit diminue d'approximativement 12%. La figure 8.3 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 section suivante analyse plus finement l'influence de la charge des processeurs sur les performances de routage.

8.1.2 Influence de la charge du processeur sur la communication

La figure 8.4 montre l'influence de la charge du processeur sur la performance en 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, remarquons que le débit de bout en bout n'est pas notablement affecté lorsque la charge des processeurs augmente.

Ces observations ne sont que partielles, et doivent être poursuivies. Deux liens seulement ont été utilisés, et pas dans les deux directions simultanément. Le comportement observé peut varier si tous les liens sont utilisés simultanément, comme le montre l'expérience suivante.

8.1.3 Influence du routage sur les programmes utilisateurs

L'expérience suivante vise 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, ils subissent déjà une baisse de 29% du temps processeur qui leur serait accordé sans le noyau. Le processus application utilisé pour cette expérience n'effectuait pas de communications. Ce résultat, constaté pour l'utilisation d'un seul lien à plein régime, nous fait craindre qu'il soit à multiplier par le nombre de liens effectivement utilisés ($4 \times 29\%$?). Il conforte l'estimation théorique effectuée en 4.2.1.3, à propos de l'équilibre des traitements et des communications sur les T800 et T414. Enfin, le processus application s'exécutait en haute priorité, et un processus de basse priorité subirait sans aucun doute une baisse plus importante.

8.1.4 Mesure d'une version plus récente

Nous avons déjà expliqué pourquoi les performances présentées dans les paragraphes précédents ne pouvaient se comparer à d'autres mesures effectuées pour d'autres logiciels de communication avec d'autres compilateurs. Notre objectif était de comprendre le comportement du noyau, et non de montrer des performances optimales. Il est cependant important d'avoir une idée des performances de ce noyau, même dans cette version.

Nous avons effectué des mesures de performances du noyau dans sa version pour C parallèle INMOS. Nous avons cette fois-ci mesuré

un autre protocole, plus lourd que le protocole de copie mémoire à distance. Il s'agit du protocole de rendez-vous, qui demande en plus l'établissement d'un rendez-vous. Nous souhaitons démontrer que les protocoles plus complexes présentent de bonnes performances.

La figure 8.5 illustre les résultats de la mesure. Le test comporte deux processus échangeant des messages en se synchronisant. Aucune charge n'est appliquée aux processeurs, et la mesure est effectuée pour un nombre variable de processeurs intermédiaires.

Les résultats obtenus se placent dans la moyenne des logiciels de communication existants. Il faut souligner que le noyau de communication présente des caractéristiques tout à fait particulières, que les autres logiciels n'offrent pas. Il permet de supporter plusieurs protocoles simultanément, et offre la garantie de l'absence d'interblocage, entre autres (voir [Gonza91]).

8.2 Mesure de la charge d'un processeur

La mesure de la charge d'un processeur est assez immédiate. Nous avons choisi d'exécuter un processus isolé une première fois, de mesurer son temps d'exécution, puis d'effectuer la même mesure avec d'autres processus s'exécutant sur le même processeur. Nous divisons alors la première valeur par la seconde pour obtenir le pourcentage de temps processeur disponible sous la charge des processus ajoutés.

Le processus de mesure utilisé 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 plus précise, ce processus peut être exécuté en haute priorité. En effet, il a alors accès à l'horloge de haute priorité, qui possède une meilleure résolution.

8.3 Imposition d'une charge au processeur

Notre second outil de mesure doit permettre d'imposer une charge d'une valeur connue à un processeur. Le principe de la solution adoptée pour imposer une charge l consiste, sur une période de temps T, à effectuer une attente active de q unités de temps, puis attendre w unités de temps. Nous avons alors les égalités suivantes :

$$\left.\begin{array}{l} T = q + w \\ l = \frac{q}{q+w} \end{array}\right\} \qquad (8.1)$$

Le processus effectuant cette boucle est dit processus de charge.

8.3.1 Choix des paramètres

8.3.1.1 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 écrit en occam. Ce dernier peut effectuer 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 de très nombreuses 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.

8.3.1.2 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 réel (ce sont les processus en dehors du noyau). Le quantum devrait donc correspondre à celui observable pour les applications parallèles. Cette valeur est malheureusement très variable en fonction des applications. De plus, les processus de haute priorité perturbent la formule 8.1 car, lors de leur exécution, ils peuvent allonger indéfiniment la valeur mesurée pour w. Cela conduit à une précision peu satisfaisante pour la valeur de la 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 8.6 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 8.7 illustre que les processus observés 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. Des résultats aussi bons, mais se stabilisant encore plus rapidement, ont 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 8.8 et 8.9 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 empire 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 observons 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 8.6). 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 8.10, 8.11 et 8.12 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 8.13. 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 illustré par la figure 8.14, 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%.

8.4 Conclusion

Cette étude des performances montre de bonnes perspectives pour le noyau de communication. Elle nous a permis de mieux comprendre son comportement et d'identifier les points à améliorer. La réception des messages peut être améliorée par l'utilisation d'un mécanisme d'anticipation. La recherche d'un protocole s'effectue par un parcours de liste, ce qui est coûteux et inutile. Un accès direct dans un tableau améliorerait de façon importante le temps de traitement d'un message. Enfin la gestion des files de requêtes aux processus serveurs du noyau peut largement être optimisée car chaque processus serveur utilise les mêmes procédures communes sans tenir compte de sa spécificité.

L'objectif de l'étude était d'une part de mettre en place des outils et une méthode de mesures, d'autre part de clarifier les paramètres importants d'un noyau de communication. Il existe un certain nombre d'idées inexactes à leur sujet, et il était important de les confronter à l'expérimentation. Nous avons par exemple vu que le dimensionnement des puissances des processeurs de traitement et de communication était important, que le routage pénalisait les programmes application, ou encore que les outils de mesure étaient difficiles à mettre au point, et que leur mauvaise qualité a des répercussions très importantes sur les résultats des mesures. Le champ des mesures et de l'expérimentation reste cependant largement ouvert.

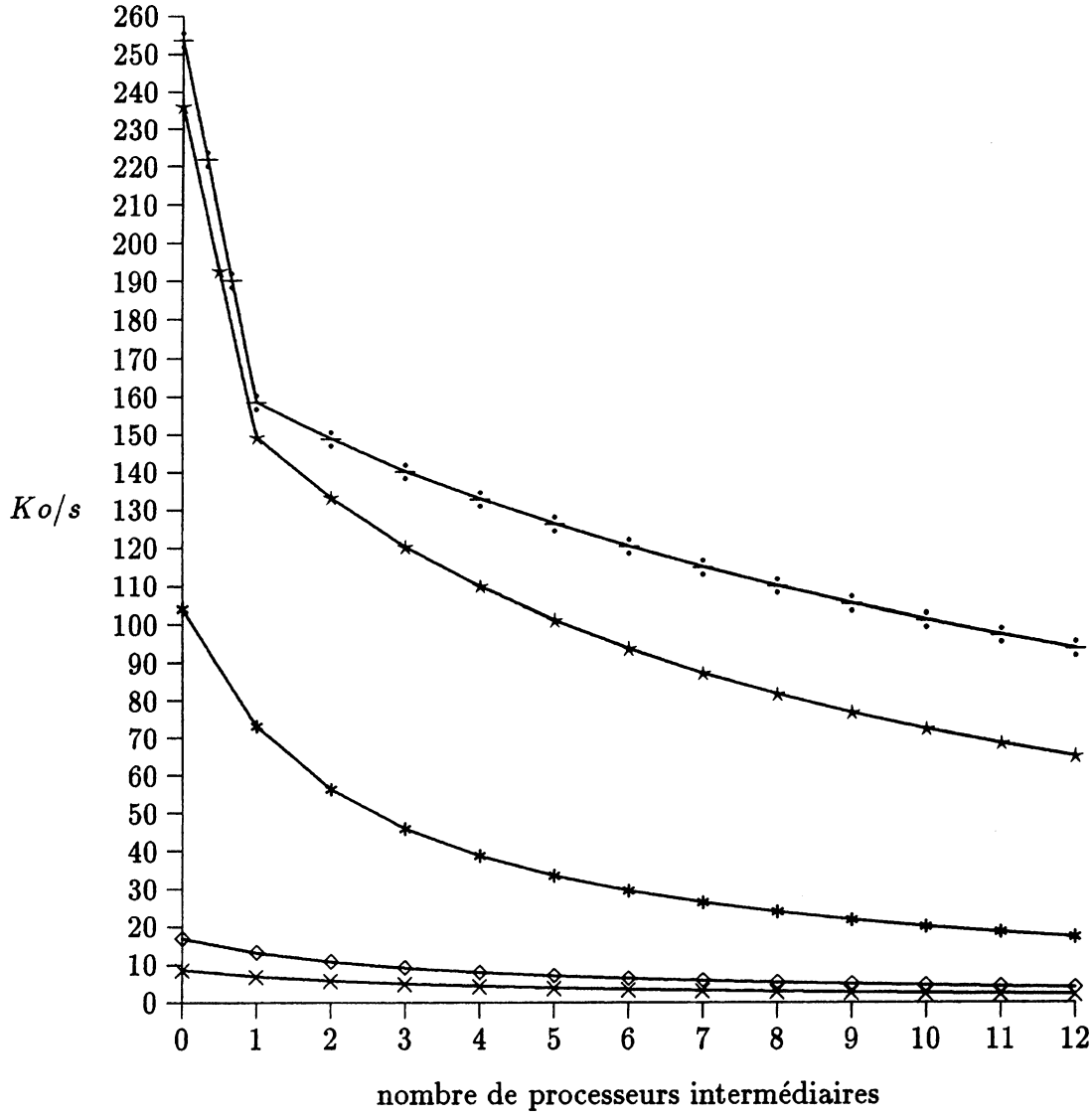


Figure 8.1: débit sous une charge uniforme de 0%

× 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

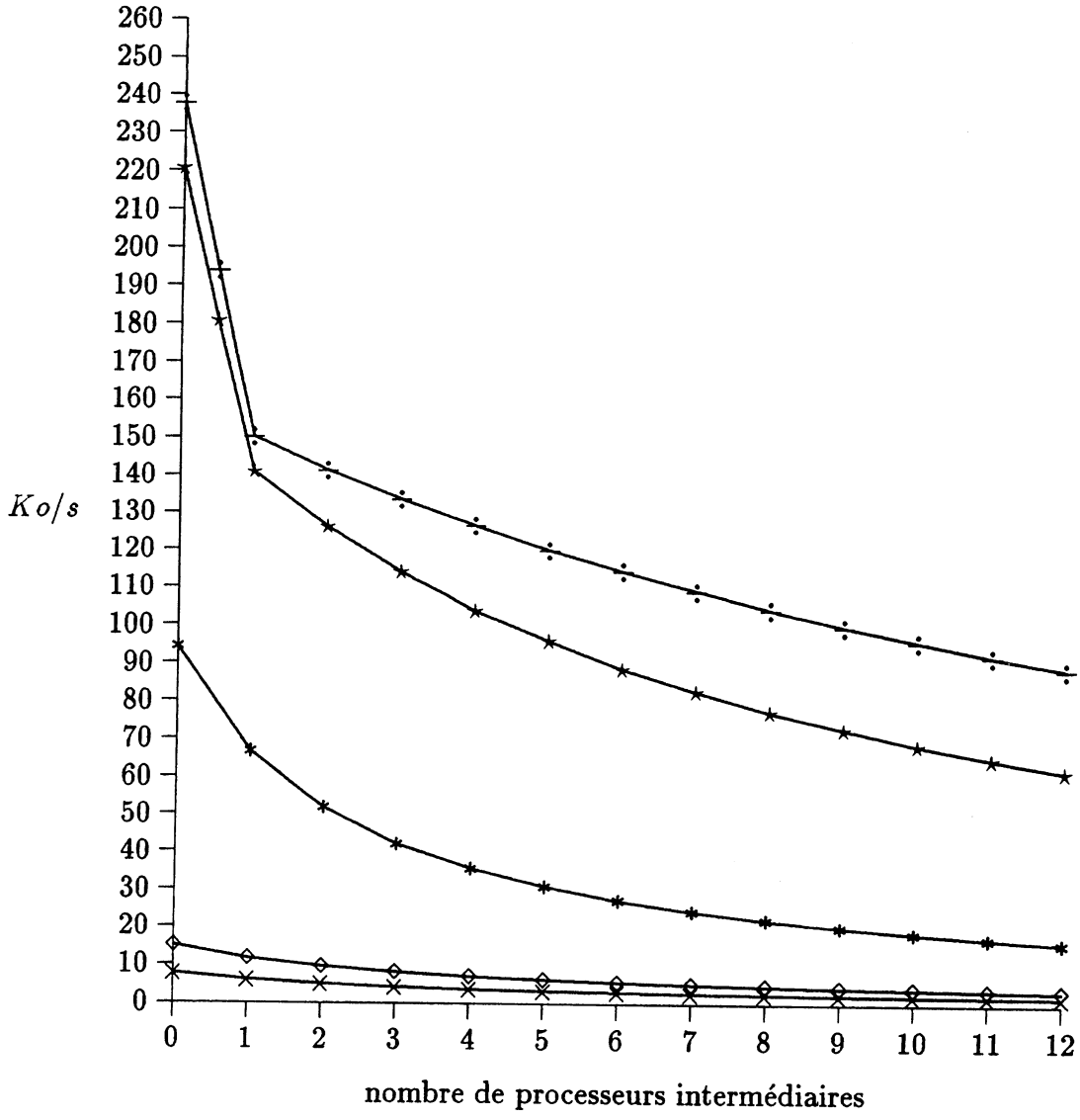


Figure 8.2 débit sous une charge uniforme de 75%

× 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

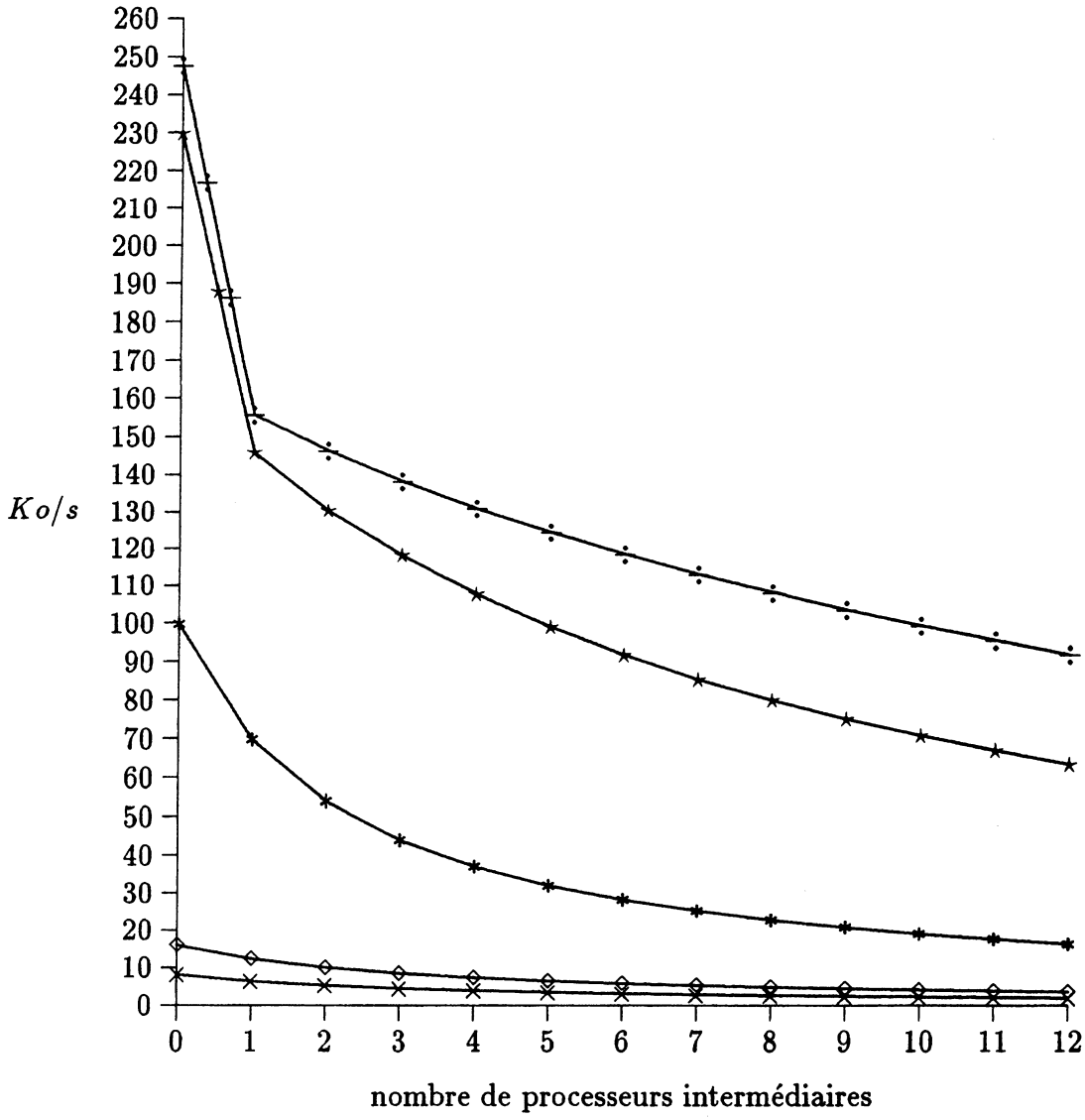


Figure 8.3: débit sous une charge uniforme de 25%

× 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

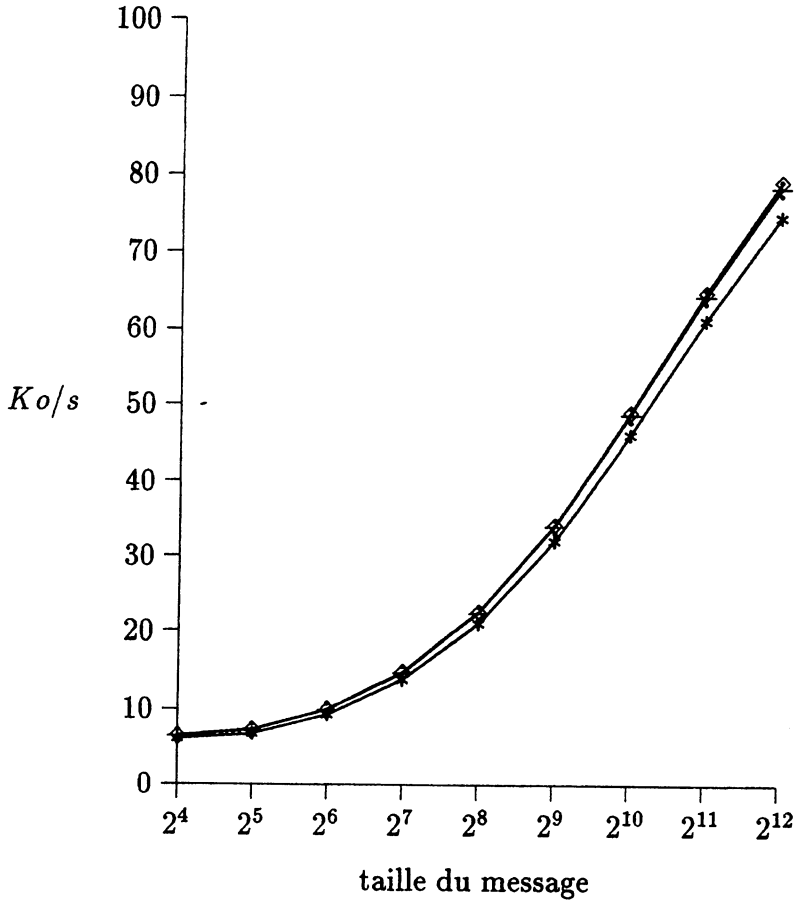


Figure 8.4: ralentissement de la communication par un processus de charge, 75, 85 et 95% de charge

÷ pour une charge de 75%

◇ pour une charge de 85%

∗ pour une charge de 95%

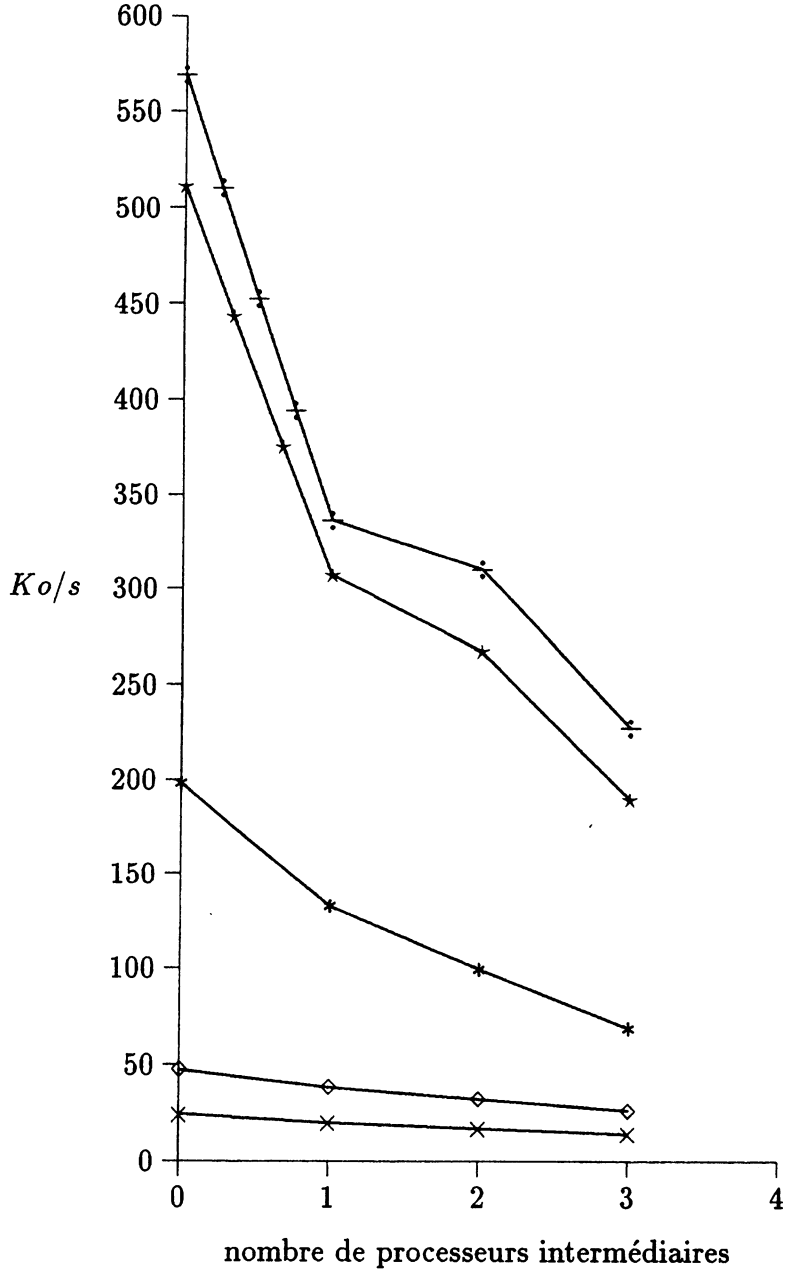


Figure 8.5: débit sous une charge uniforme de 0%, secondes mesures

× 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

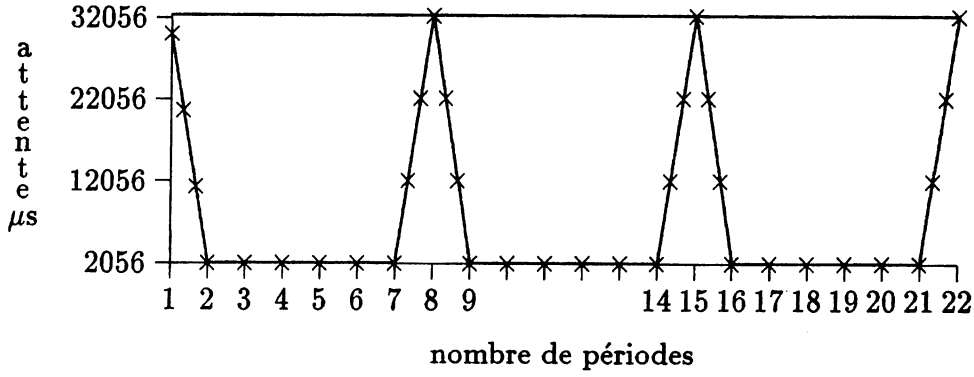


Figure 8.6: variations du temps d'attente w, pour $q = 1024\mu s$, $l = 50\%$

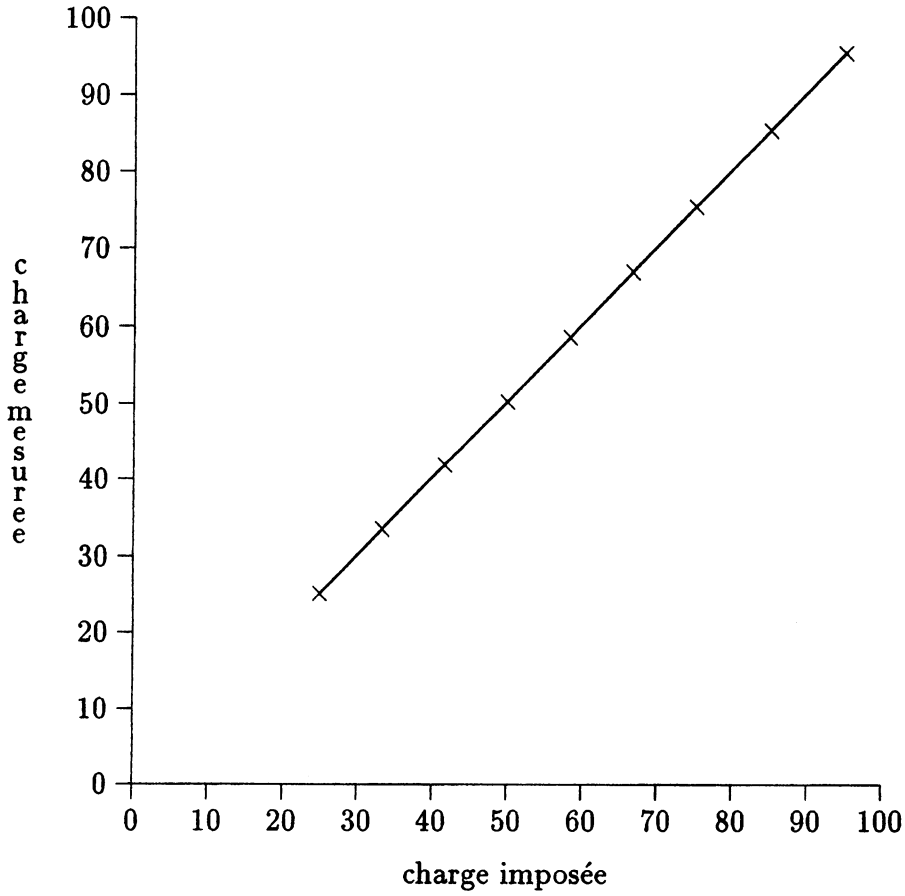


Figure 8.7: mesure de l'effet de la charge imposée sur les processus de basse priorité, $q = 1024\mu s$, $l = 25, 50, 75\%$

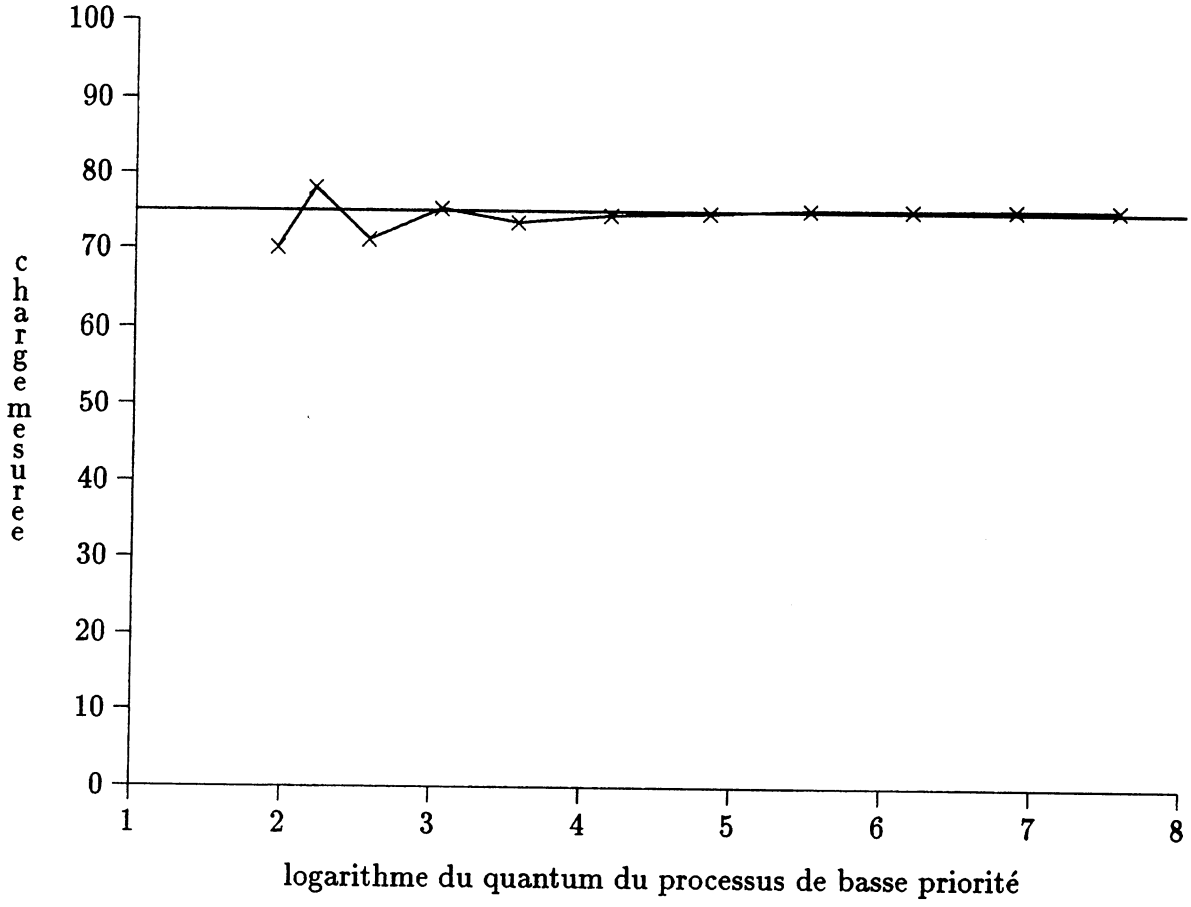


Figure 8.8: effet de la charge imposée pour différentes valeurs du quantum du processus de basse priorité mesuré, $q = 1024\mu s$, $l = 75\%$

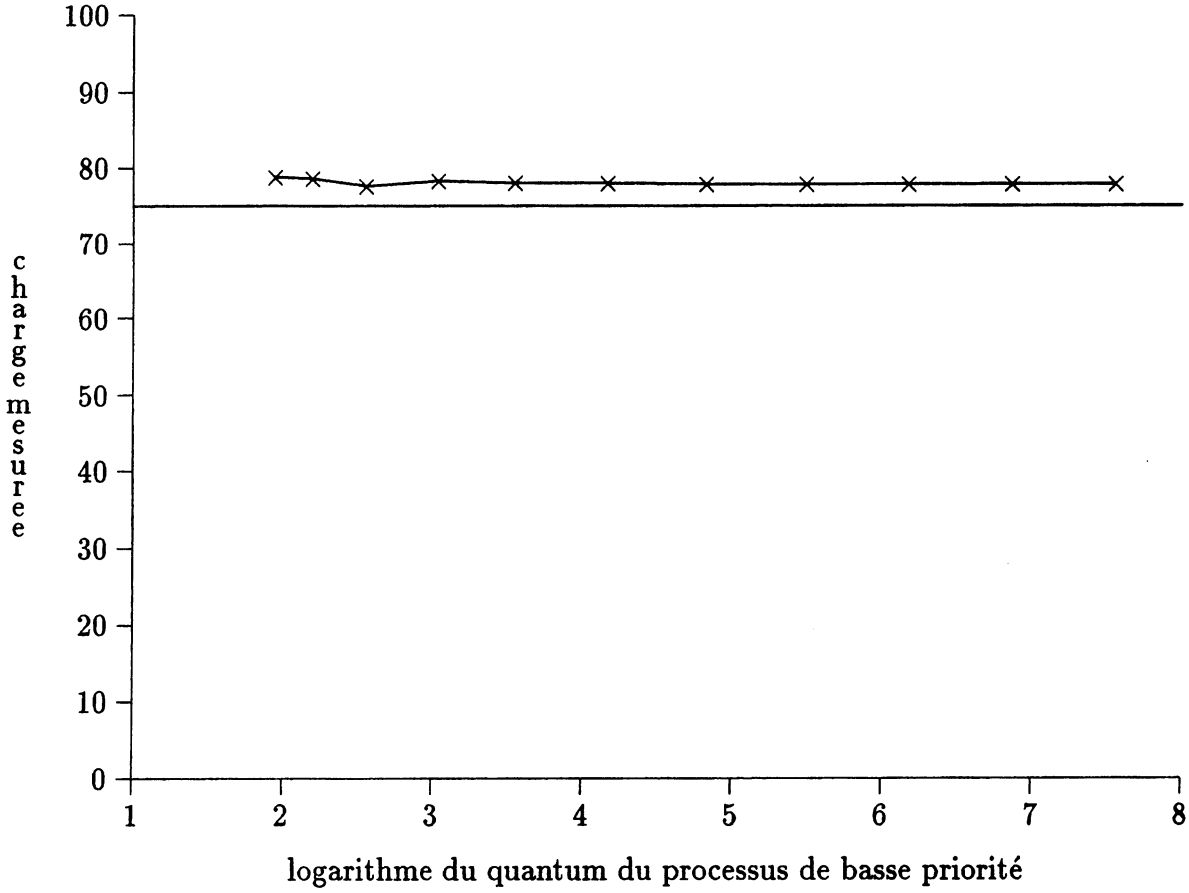


Figure 8.9: effet de la charge imposée pour différentes valeurs du quantum du processus de basse priorité mesuré, $q = 128\mu s$, $l = 75\%$

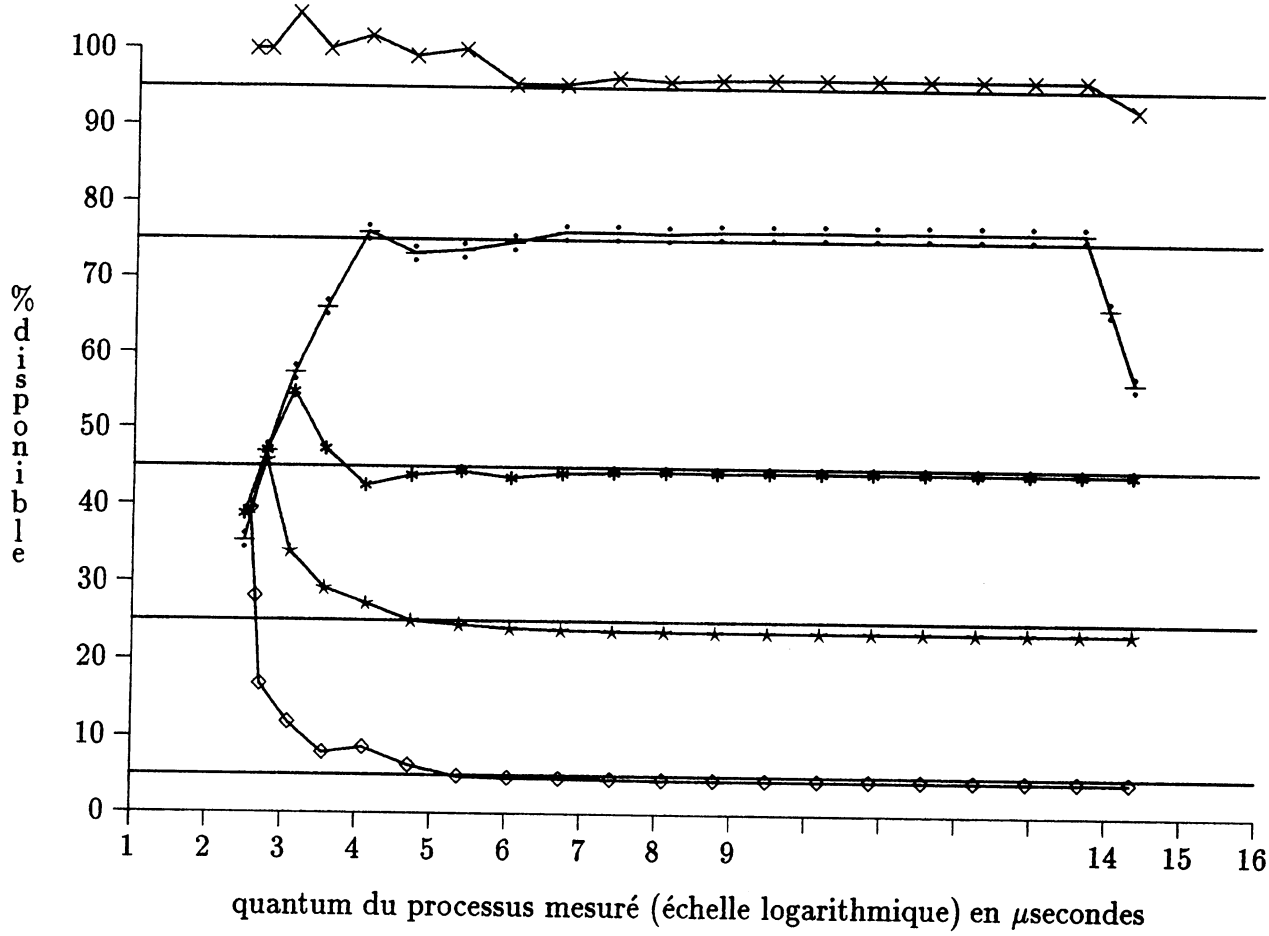


Figure 8.10: pourcentage de processeur disponible aux processus de haute priorité en fonction du quantum du processus observé, 5, 25, 45, 75 et 95%

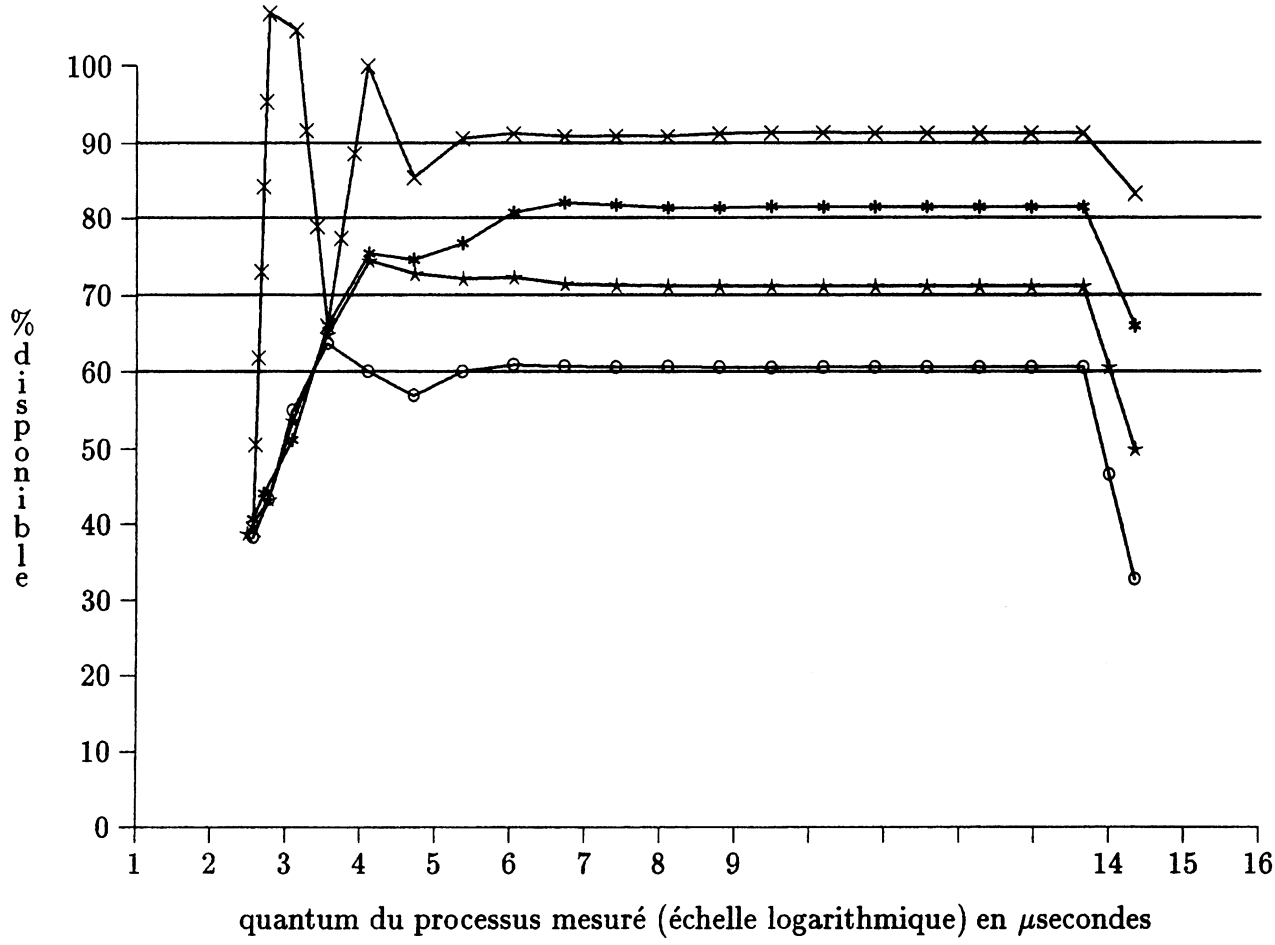


Figure 8.11: pourcentage de processeur disponible aux processus de haute priorité en fonction du quantum du processus observé, 60, 70, 80 et 90%

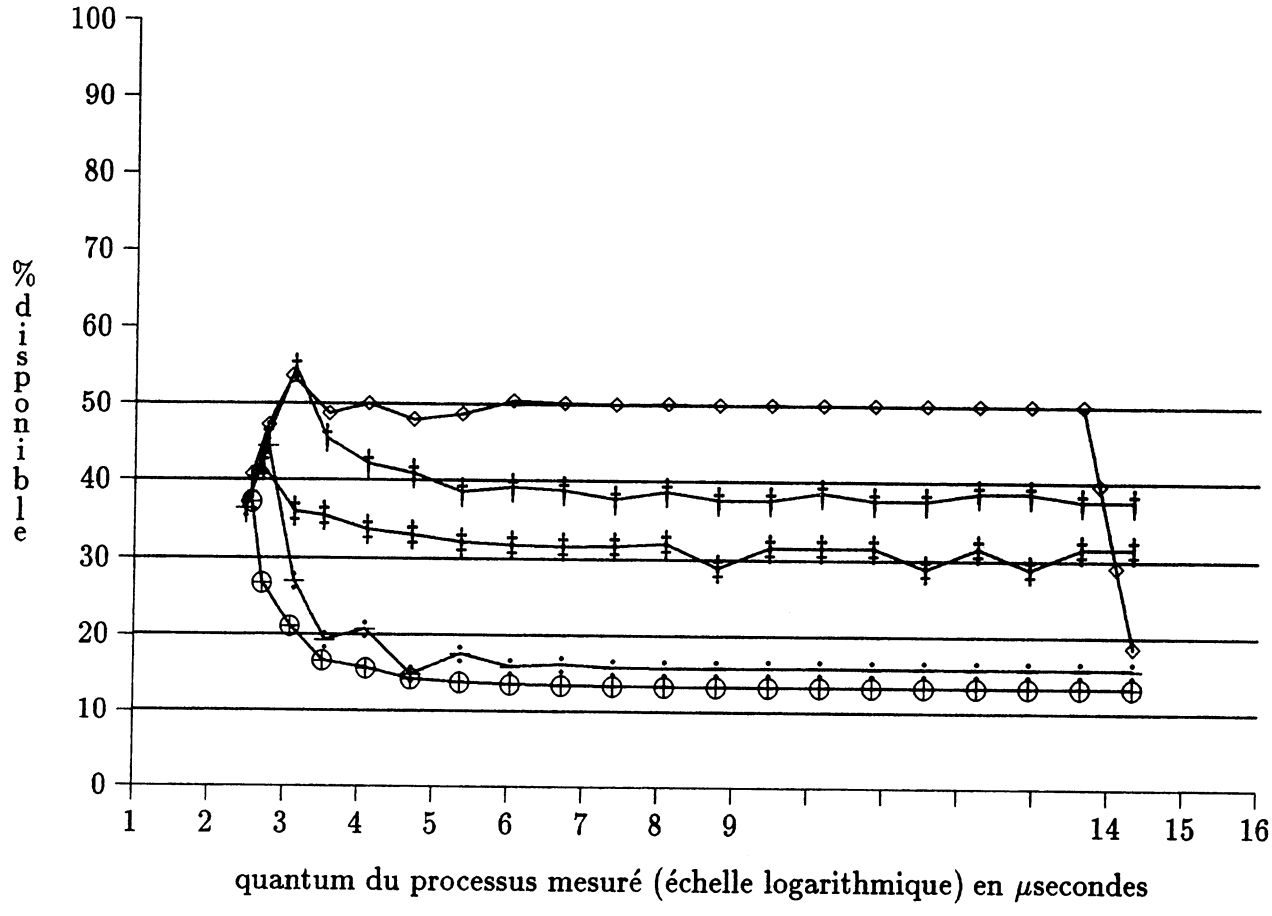


Figure 8.12: pourcentage de processeur disponible aux processus de haute priorité en fonction du quantum du processus observé, 10, 20, 30, 40 et 50%

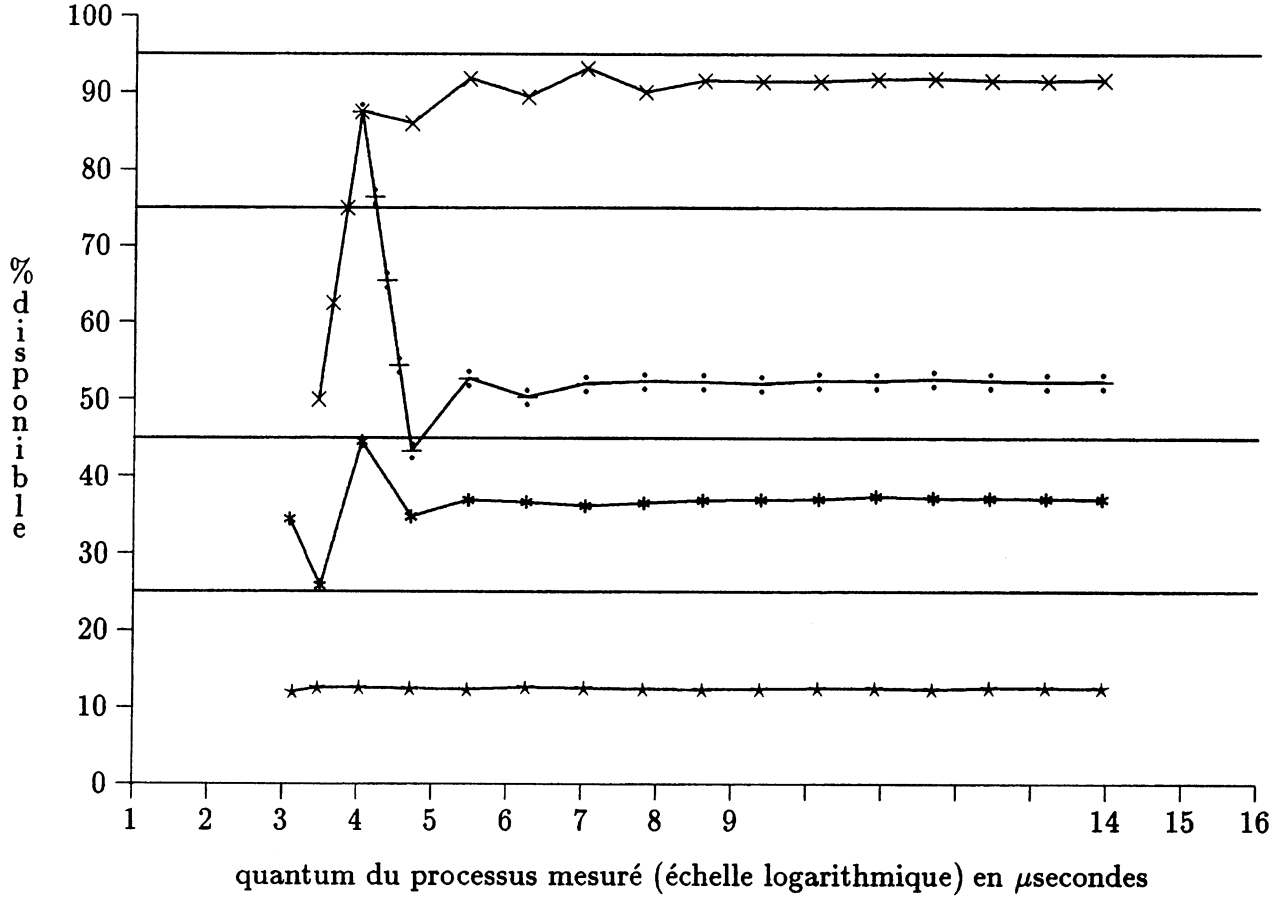


Figure 8.13: pourcentage de processeur disponible aux processus de basse priorité en fonction du quantum du processus observé, 25, 45, 75 et 95%

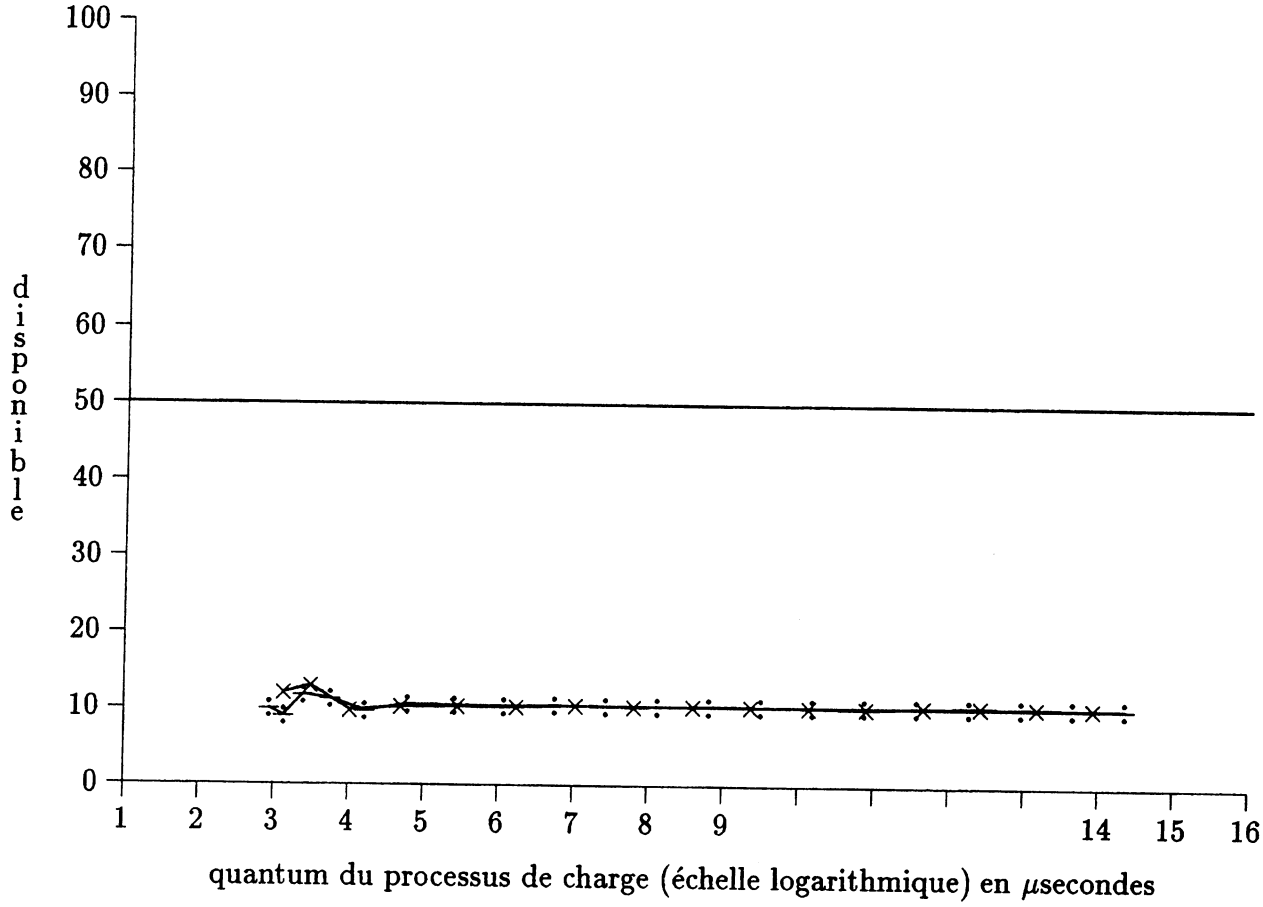


Figure 8.14: charge mesurée pour un processus de basse priorité, en faisant varier le quantum du processus de charge, $l = 50\%$

Chapitre 9

Autres systèmes parallèles : comparaison

Nous revoyons ici les principaux choix de Parx et les comparons à d'autres grands projets sur les systèmes parallèles. La large majorité de ces projets sont européens, les américains se concentrant sur les systèmes multi-processeurs et les systèmes distribués. Notons tout de même que Trollius [Braner88] et TopExpress proviennent d'outre Atlantique.

Cette situation est en train de changer, Cray parle de 16 processeurs en 1991 (Cray III), et certains parlent de 64 processeurs en 1995 (Cray IV) [OAK90]. Intel, poursuivant sa série IPSC, introduit le iPSC/860, à base du processeur i860, qui supporte de 8 à 128 processeurs comme ses prédecesseurs. De même, les Japonais ne sont pas en reste. En conclusion de la seconde conférence "Supercomputing Europe'90", J.R. Sherman souligne que la tendance actuelle est au parallélisme pour les super-ordinateurs [RiSh90].

En Europe, les principaux projets dans les systèmes à haut degré de parallélisme, sont le projet ESPRIT "European Declarative System" [IsBor90,BBH89], Genesis [WSP90], basé sur le projet allemand "Suprenum", et le trio de projets ESPRIT Supernode II, PUMA et GPMIMD.

Plusieurs industriels commercialisent des systèmes d'exploitation pour machines parallèles, notamment Helios [Gar87], Idriss [King88] ou Meikos. Nous ne nous intéresserons ici qu'à Helios, qui semble le plus avancé du point de vue du parallélisme.

9.1 Présentation des systèmes

Les projets EDS et Suprenum présentent des architectures de systèmes très semblables à Supernode. Les systèmes sont bâtis à partir

d'un modèle de processus communiquants.

EDS développe PCL (Process Control Langage) comme interface au noyau de système. Suprenum propose PEACE (Process Execution And Communication Environment), un système distribué pour machines à haut degré de parallélisme.

9.1.1 EDS

Le projet EDS vise à la réalisation d'un système de développement extensible pour multiprocesseurs. L'une des idées de base est l'extensibilité du système. Les applications visées sont des serveurs parallèles de bases de données relationnelles, et des applications écrites en Lisp et Prolog parallèles.

La machine cible est un réseau pouvant comporter jusqu'à 256 nœuds, chacun disposant d'une mémoire privée importante. Les nœuds sont connectés suivant un réseau delta.

L'objectif pour le noyau du système d'exploitation est de fournir un support pour le parallélisme et de supporter différents modèles d'exécution, en particulier d'être multitâches et multi-utilisateurs. Constatant l'inadéquation des noyaux de systèmes existants, EDS décide de concevoir un noyau de système nouveau, tout en conservant des services systèmes traditionnels, dérivés d'Unix. Un aspect important du noyau est la mise en œuvre d'une mémoire virtuelle partagée.

9.1.2 Suprenum

Le projet allemand Suprenum a pour objectif le développement d'un super-ordinateur. Peace est un système d'exploitation distribué pour la machine Suprenum. Les fonctionnalités essentielles de Peace sont la gestion des processus et la communication par échanges de messages. Les applications visées sont les candidates habituelles des super-ordinateurs, les applications numériques.

9.1.3 Helios

Helios est un système d'exploitation destiné à des réseaux maillés de processeurs. Il fut développé à l'origine pour le transputer, mais il permet aujourd'hui d'intégrer d'autres types de processeurs dans un même réseau.

Il s'agit à la base d'un environnement de développement permettant de piloter un réseau de processeurs à partir d'une machine hôte. Les

applications visées sont celles dérivées du monde Unix, et des applications numériques.

Helios propose des fonctions de communication et un modèle de processus ayant pour objectif de se rapprocher le plus possible de la compatibilité Unix. A sa conception, il n'était pas associé à un projet de recherche et développement à long terme mais répondait à une demande du marché des transputers.

9.2 Architectures de systèmes

9.2.1 Pcl

EDS a choisi Chorus [AGHR89] comme système de départ. Le noyau assure les fonctions de communication entre processus. Dans le but de supporter les modèles de programmation par données partagées, EDS propose une mémoire virtuelle partagée par tous les processeurs. La machine cible est un réseau de "Processor Element"s ou PEs. Chaque PE comporte un processeur de gestion de mémoire virtuelle et une mémoire locale. Il n'y a pas de mémoire partagée entre PEs.

Soucieux de ne pas restreindre les possibilités d'utilisation de PCL, le noyau implante des mécanismes, laissant le soin aux couches supérieures de mettre en œuvre les politiques correspondant à leurs applications propres.

9.2.2 Peace

Il procède de la même démarche. La machine physique est formée de "clusters" regroupant chacun des processeurs physiques. Les clusters sont reliés dans un tore à deux dimensions. Chacun dispose d'un processeur de communication permettant de le relier à quatre autres clusters par quatre liens à haut débit. Dans un cluster, la communication entre processeurs passe par un bus double. Chaque cluster possède des processeurs de travail, de communication et de service (disques etc.).

On retrouve un cas de figure que permet Parx, dans lequel la topologie du cluster est statiquement fixée en bus.

Un noyau assure les fonctions de communication entre processus. Il est conçu de façon à ne pas restreindre la gamme des applications possibles ni l'efficacité de leur implantation. Les appels systèmes sont effectués par appels de procédure à distance, afin de ne pas gaspiller de mémoire en dupliquant le code système sur tous les processeurs.

9.2.3 Helios

Il s'exécute sur des machines parallèles configurées en réseau maillé statique. A nouveau, un noyau minimal, le "nucléus" contrôle l'allocation des processeurs et de la mémoire, et permet la communication entre processus. Les autres fonctions systèmes sont assurées par des programmes systèmes s'exécutant comme des processus utilisateurs.

9.2.4 Bilan

Il y a donc un large consensus sur le fait que les systèmes parallèles ne puissent offrir un support couvrant toutes les applications qu'ils envisagent. Un noyau de système offre un support d'exécution pour des processus communicants, et éventuellement pour la gestion de la mémoire.

Peace et Parx décomposent la machine physique en parties plus petites appelées clusters, dans lesquelles un support plus spécifique peut être développé. Les clusters de Parx sont cependant des entités dynamiques, ajustables en fonction des besoins. Ils possèdent aussi une configuration sur laquelle on peut agir afin de la rapprocher de la machine idéale souhaitée. Les clusters de Parx permettent d'offrir des vues différentes de la même machine physique en offrant des environnements de programmation différents dans les clusters.

Ils distinguent aussi la puissance de traitement allouée à l'utilisateur de celle nécessaire au système d'exploitation. Enfin, un intérêt tout particulier est porté aux entrées-sorties. En effet, rien ne sert de multiplier la puissance de traitement si les processeurs ne peuvent être alimentés en programmes et données, et les résultats collectés. Seymour Cray ne disait-il pas :

> "A Supercomputer is a device that turns a compute intensive problem into an I/O intensive problem"?

9.3 Modèles 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. Comme nous l'avons vu en 6.2, plusieurs fonctionnalités sont regroupées dans le processus traditionnel. Les différents modèles de processus décrits ici correspondent à des combinaisons différentes de ces fonctionnalités.

9.3.1 Pcl

Il propose un modèle de processus à trois niveaux. L'unité d'exécution élémentaire est le "thread", qui a la même signification que dans Parx. Les threads s'exécutent dans l'espace d'adressage de "tâches". Une tâche représente un espace virtuel d'adressage, pouvant s'étendre sur plusieurs PEs, ce qui n'est pas le cas dans Parx. Pcl introduit à nouveau un modèle à mémoire partagée entre clusters. On peut se demander quelle est l'utilité de la structure de PE si elle ne constitue pas une borne à l'espace d'adressage d'une tâche. Certains threads d'une tâche multi-PEs s'exécutent en parallèle. L'ensemble des threads d'une même tâche s'exécutant sur le même PE est appelé "team". Un team représente un espace de mémoire physique partagé entre plusieurs threads qui s'exécutent en pseudo-parallélisme, ou en parallèle si le PE est multi-processeur.

Enfin, un "job" est une unité administrative, représentant un utilisateur et regroupant plusieurs tâches. Il n'y a pas de protection au sein d'un job.

Les tâches permettent de proposer un modèle de parallélisme avec données partagées, tout en supportant simultanément un modèle par échange de messages. Rappelons que EDS ne supporte pas la mémoire partagée entre PEs.

Les jobs sont ordonnancés par tranches de temps, ainsi que les différents teams d'un job. Les threads d'un team ont chacun une priorité, et c'est le thread de priorité la plus élevée qui s'exécute, avec préemption au besoin. Un mécanisme d'élevation temporaire de priorité est disponible pour réaliser l'exclusion mutuelle.

Le modèle de Parx comporte le même nombre de niveaux avec des fonctionnalités semblables. Il nous semble plus simple et plus clair que celui de PCL. Ce dernier offre des supports redondants, par exemple pourquoi placer des threads sur des PE 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 PEs ?

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. PCL offre des primitives pour valider des écritures, ce qui synchronise les multiples copies d'une même page physique.

L'ordonnancement offre plus de souplesse dans le choix des priorités et des quanta que Parx. Cependant, cette politique n'est pas cohérente avec le concept de tâche.

9.3.2 Peace

Il établit un parallèle entre un programme parallèle et une fédération de foot-ball. Un "thread" est un flot de contrôle d'instructions, encore une fois exactement comme dans Parx. Un "team" représente un espace d'adressage pour plusieurs threads, physiquement isolé d'autres espaces. Rappelons que les processeurs ne partagent pas de mémoire. Un team est donc localisé à un processeur.

Un programme est représenté par une "entité", qui regroupe ses différentes fonctionalités. Celles-ci sont l'application elle-même, la machine abstraite implantant les constructions du langage, le "run-time" système, la bibliothèque système et le couplage de services (support pour appels de procédures à distance). Un team peut contenir une ou plusieurs entités. Enfin, une application distribuée peut regrouper plusieurs teams, elle forme alors une "league".

A nouveau, nous retrouvons un modèle de processus à trois niveaux. Cette fois-ci, nous sommes en mémoire non partagée, et chaque niveau a un correspondant dans Parx. Le code des programmes systèmes est accédé par appels de procédure à distance, alors que dans Parx, il est exécuté par des serveurs auxquels les utilisateurs envoient des messages.

9.3.3 Helios

Il permet à un programme utilisateur, une "task force", de se décomposer en "tâches". Une task force est décrite par un fichier de configuration, qui exprime les relations entre tâches. Une tâche peut, à son tour, contenir des processus légers appelés "processus".

La task force n'a pas d'existence propre, elle peut être considérée comme un ensemble de commandes utilisateur pour créer un réseau de tâches communicantes. La tâche est l'unité administrative et de distribution. Elle représente un espace d'adressage. Les processus se déroulent en pseudo-parallèle au sein de tâches. Il n'y a pas d'ordonnacement spécifique par tâche.

Ici, nous avons un modèle de processus à deux niveaux, le troisième étant offert par l'interface de programmation par un langage de description le "Common Description Langage". Le modèle de processus de Parx est incontestablement plus riche, plus complet et plus cohérent.

9.4 Modèles de communication

Bien que les systèmes reconnaissent ne pas pouvoir offrir tous les protocoles dont auraient besoin les applications, ils 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.

9.4.1 Pcl

Il propose une communication par ports. Un port est une file d'attente de messages, de taille quelconque, qui peuvent être lus par un team. Chaque port a un nom global, plusieurs extrêmités d'émission, une de réception. L'émetteur fournit avec son message un identificateur permettant au serveur de lui répondre.

L'utilisateur associe un pool de buffers au noyau de communication pour les ports. Le contrôle de flux est effectué par l'application. Trois indicateurs du niveau de remplissage du pool sont disponibles. Au passage de l'un d'entre eux, une exception est générée pour l'utilisateur, qui peut alors engager une action correctrice. Ces trois niveaux sont les remplissages maximum, normal et bas du pool. L'utilisateur peut dynamiquement rajouter ou retirer des buffers. Si un message dépasse la taille du pool, il est délivré de façon synchrone à l'utilisateur qui est responsable d'allouer de la mémoire pour lui.

La communication peut être synchrone ou asynchrone. Dans le premier cas, l'émetteur est certain que le message est dans le pool du récepteur. Un protocole d'appel de procédure est aussi disponible. L'émetteur attend alors une réponse du team récepteur.

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 de pages mémoires. Les pages peuvent être lues et écrites indépendamment de leur exemplaire original, 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 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 buffers comme PCL.

9.4.2 Peace

Il propose trois protocoles de communication, qui sont en fait des variations du même protocole de base. Ils utilisent tous des ports. Un port identifie un processus de façon unique dans le système. La communication est synchrone, sans buffers.

Le premier est un appel de procédure à distance. 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.

Le second est destiné aux échanges de gros volumes d'information. L'émetteur désigne un processus destinataire, un couple d'adresses source et destination, et une taille. L'émetteur doit auparavant se synchroniser avec le récepteur, en utilisant le protocole déjà décrit. Le transfert a ensuite lieu, lors de sa terminaison l'émetteur est débloqué et il peut envoyer une réponse au récepteur, toujours à l'aide du premier protocole. Un schéma symétrique, demandant un message supplémentaire, permet de lire de l'information.

Le troisième et dernier est en fait une combinaison des deux premiers. 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.

Les protocoles proposés ici sont couverts par Parx à travers deux protocoles différents (accès direct mémoire et client-serveur). Peace adopte la même approche sans buffers. Il 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. Des essais de migration ont été effectués, mais une telle décision n'est pas du ressort du noyau. Il se contente de fournir un ensemble de mécanismes permettant la migration.

9.4.3 Helios

Helios propose une communication par ports. 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" permet-

tent de relayer des messages de processeur en processeur jusqu'à leur destination finale.

Le système ne garantit pas l'acheminement des messages, et fournit un mécanisme de délai de garde pour se prémunir contre les pertes.

Le modèle de communication de Helios 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.

9.5 Conclusion

Nous pouvons conclure que Parx, couvrant à bas niveau les protocoles de communication de tous les systèmes présentés, est le mieux conçu du point de vue de la communication. EDS est conçu pour supporter la programmation fonctionnelle et logique à travers les langages Prolog et Lisp, et aussi une grosse base de données. Suprenum vise un large domaine d'applications, numériques entre autres. Helios se veut un système distribué pour station de travail dotée d'un accélérateur de calcul.

Ceci laisse penser que Parx peut être comparé favorablement à ces différents projets. Les applications étudiées incluent la programmation logique, les applications numériques, la simulation, les bases de données, les programmes parallèles et les environnements plus classiques.

Partie III

Conclusion

Chapitre 10

Conclusion et perspectives

Au terme de ce travail, et à la lumière des évolutions des machines et systèmes informatiques, nous pouvons affirmer que les systèmes parallèles sont en plein développement et promis à un avenir certain. Ceci est confirmé par le changement d'orientation que l'on peut noter dans l'industrie et la recherche américaines, qui adoptent aujourd'hui une approche par le parallélisme, après de nombreuses années consacrées aux machines vectorielles.

Plusieurs chemins sont possibles dans la voie du parallélisme, et nous les avons présentés au cours de cet ouvrage. Dans tous, il apparaît qu'un programme parallèle est un tout cohérent qui demande un support spécifique, suffisamment flexible pour s'adapter aux besoins du programme. Il se dégage la nécessité de sous-systèmes qui répondent chacun aux besoins de sous-ensembles d'applications.

L'architecture de Parx répond parfaitement à cet objectif de flexibilité et de cohérence. Le modèle d'exécution proposé, les tâches parallèles, tâches et threads, est suffisamment général pour répondre aux besoins des applications parallèles et des applications des systèmes distribués actuels.

Les modèles de programmation, qui sous-tendent les applications parallèles, mettent en œuvre de multiples entités pour le parallélisme, et plusieurs paradigmes de communication. Nous avons proposé de construire les divers protocoles de communication à partir d'une base commune correcte. Le noyau de communication intègre cette base de protocoles de façon suffisamment flexible pour permettre l'ajout d'autres protocoles, et la modification dynamique de leur comportement.

Chaque application parallèle définit une architecture cible "idéale". Celle-ci peut évoluer au cours de l'exécution du programme. Mais comment déterminer la meilleure architecture pour le cas général ? Cette question n'a pas encore trouvé de réponse, et aucune ne semble émerger. D'où la nécessité d'une très grande souplesse au niveau de la machine virtuelle offerte par le système d'exploitation.

A cette fin, les systèmes d'exploitation actuels sont structurés autour de noyaux "minimaux" qui supportent différents sous-systèmes permettant d'offrir de multiples modèles de programmation.

Il est intéressant de noter que les applications traditionnelles, non parallélisées, tirent aussi avantage de cette structuration. Ceci est confirmé par les tendances des systèmes distribués.

La première ébauche du système Parx remonte aujourd'hui à quatre ans. Elle a été développée et améliorée depuis, et nous avons pu prendre en compte des idées nouvelles ainsi que la critique de nos collègues, des partenaires du projet Supernode et l'avis de la communauté scientifique. Parx aujourd'hui, dans ses aspects essentiels, correspond à ce que nous souhaitions réaliser. Il y a évidemment certains aspects importants qu'il aurait été intéressant de développer.

Il existe une multitude de champs de recherche ouverts ou réactualisés avec le parallélisme. Ils concernent l'expression des volontés du programmeur dans les langages. Cette expression doit être agréable (interface homme-machine), puissante et efficace. Une approche complémentaire consiste à extraire des informations de programmes. Une étape de projection sur une machine virtuelle est ensuite nécessaire, qui doit bien se garder de trahir les intentions du programmeur, tout en faisant le meilleur usage possible de la machine physique. Les langages de programmation, leur traduction intelligente, la gestion d'entités représentant des concepts abstraits du programmeur (données distribuées) et la gestion des ressources système ont chacun des résultats à apporter dans cette entreprise.

Appendix A

Un problème d'initialisation

Nous présentons ici le problème de l'initialisation d'un réseau de transputers. Nous insistons sur la diffusion des tables de routage lors de l'initialisation d'un réseau. Cette initialisation peut se faire de deux manières : par une mémoire de type "ROM" se trouvant sur chaque processeur, ou bien par l'intermédiaire d'un lien de communication, l'option qui nous intéresse ici.

Cette présentation ne prétend pas aborder une question fondamentale ou difficile. Il s'agit simplement de faire percevoir le type de problème que nous avons eu à résoudre tout au long de notre travail de développement.

A.1 Présentation

Le réseau de transputers forme un graphe connexe, et un ou plusieurs liens physiques permettent d'accéder à un support de mémoire contenant l'image mémoire initiale des processeurs à partir d'un processeur appelé "racine" (voir figure A.1). Le chargement des processus s'effectue de proche en proche à partir de la racine, ce qui définit un arbre de chargement.

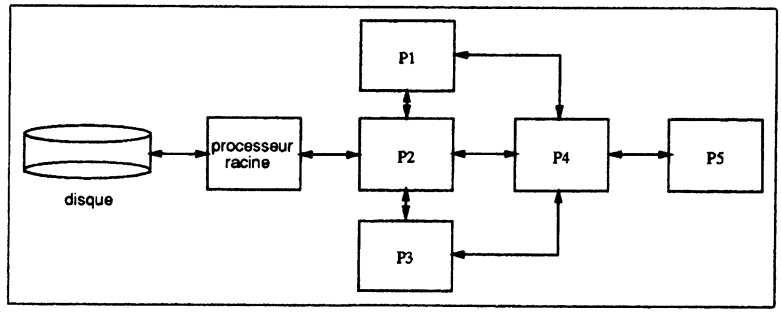


Figure A.1: Exemple de réseau de transputers

Il s'agit de donner à chaque processeur son image mémoire. Celles-ci diffèrent (en particulier) par la table de routage associée à chaque processeur. L'initialisation d'un transputer s'effectue en deux étapes :

1. envoi d'une séquence d'octets (255 au plus) contenant un programme initial à exécuter,
2. le programme initial charge un programme complet pour le processeur.

Dans le cadre de Parx, il s'agit de charger le noyau du système sur un réseau quelconque de processeurs. Nous procédons pour cela en quatre étapes :

1. initialisation de chacun des transputers du réseau. Etablir une synchronisation de façon à s'assurer que tous les processeurs sont initialisés, et donc que l'on peut leur envoyer des messages.
2. chargement du noyau sur chacun des transputers. Etablir une synchronisation pour s'assurer que le chargement est terminé. Le programme du noyau est identique pour tous les processeurs.
3. envoi des tables de routage. Chaque processeur possède sa propre table. Synchronisation afin de s'assurer que tous les processeurs ont reçu leur table.
4. lancement du noyau sur chacun des processeurs.

Les étapes 2 et 3 sont distinguées parce que dans le premier cas, il suffit de dupliquer un même programme sur tous les processeurs, alors que dans le second, il faut chaque fois lire l'information à partir du processeur racine.

Ce problème peut être résolu de différentes manières. Nous choisissons d'utiliser le maximum de parallélisme et d'éviter les copies inutiles. Une première solution consiste à envoyer, suivant l'arbre de chargement, toutes les tables avec l'image du noyau, et ensuite effectuer une sélection en fonction du numéro de processeur. Elle ne nous intéresse pas, car :

- elle entraîne des recopies inutiles. En effet, il faut au total copier $d_r \times N^2 \times S$ octets, d_r étant la distance moyenne de la racine à un processeur, N le nombre total de processeurs, S la taille de la table de routage d'un processeur (exprimée en octets). Le nombre minimal de copies nécessaires est $d_r \times N \times S$.
- il faut ensuite, sur chaque processeur, sélectionner sa table et l'extraire (ce qui demande une nouvelle copie) de la table globale, afin de récupérer la place occupée par la table globale.

Nous souhaitons paralléliser l'initialisation du réseau. Nous retenons le principe de commencer une étape aussitôt que possible.

A.2 Algorithme non parallélisé

A.2.1 Phase d'initialisation

Nous supposons que le processeur racine est initialisé par une procédure quelconque (machine auxilliaire d'initialisation). Il a accès à un système de fichiers. Il exécute le programme de la figure A.2. Il consiste à lire les instructions du programme d'initialisation, puis ses données (la description de l'arbre de chargement), et à les envoyer vers le processeur relié à la racine. Il faut ensuite attendre la fin de la propagation du programme.

```
void boot(boot_file, tree_file)
int boot_file, tree_file; /* descripteurs de fichiers */
/*
  boot_file : fichier contenant le programme d'initialisation
  tree_file : fichier contenant l'arbre de chargement
*/
{
  uchar boot_code[BOOT_SIZE]; /* instructions du programme */
  uchar boot_tree[TREE_SIZE]; /* 4 bits par processeur */
  uchar boot_sequence = BOOT_SIZE + TREE_SIZE;
  int ACK;  /* message de synchronisation */
  read(boot_file, boot_code, BOOT_SIZE);
  read(tree_file, boot_tree, TREE_SIZE);
  put_block(sizeof(boot_sequence), &boot_sequence);  /* taille sequence */
  put_block(BOOT_SIZE, boot_code);  /* envoi du code de boot */
  put_block(TREE_SIZE, boot_tree);  /* envoi des donnees du boot */
  /* synchronisation sur la fin de premiere etape */
  get_block(sizeof(ACK), &ACK);
  return;
}
```

Figure A.2: Phase d'initialisation (racine)

Le programme d'initialisation chargé est illustré à la figure A.3. Il se résume à l'initialisation du processeur, à la numérotation au vol des processeurs du réseau et au chargement des processeurs fils dans l'arbre de chargement. Il faut attendre la fin du chargement de chaque branche d'arbre correspondant à un fils, puis envoyer un compte-rendu de terminaison au processeur père.

A.2.2 Phase de chargement du noyau

Lors de cette phase, le processeur racine exécute le programme illustré à la figure A.4. C'est simplement le chargement du programme du noyau sur le processeur relié à la racine. Il faut ensuite attendre la fin du chargement de tout le réseau.

```
int numero = 1; /* variable passee de proc. a proc. pour numerotation */
int numero_prive; /* numero du processeur local */
uchar children[TREE_SIZE];
int bootlink;   /* lien par lequel j'ai ete initialise */
int ACK;  /* accuse de reception */
void boot(void)
{
  int i;
  /* initialisation des horloges, files de processus, canaux,
     calcul de la taille memoire etc, NON DETAILLE.
  */
  numero_prive = numero;
  numero += 1;  /* numero du processeur suivant */
  ACK = numero; /* que l'on retourne dans l'accuse de reception */
  for (i = 0; i < MAX_CHILDREN; i++)
  {
    if ((children[numero_prive])[i] == IS_TO_BOOT)
    { /* le processeur correspondant au lien i doit etre initialise */
      boot_processor(i);  /* se cloner sur le processeur fils */
      in(i, &ACK, sizeof(ACK)); /* attendre fin de chargement sous-arbre */
      numero = ACK; /* numero de processeur suivant */
    }
  }
  /* envoi de compte rendu au pere */
  out(boot_link, ACK, sizeof(ACK)); /* pour permettre la synchronisation */
}
```

Figure A.3: Phase d'initialisation (réseau)

Chaque processeur exécute alors le programme illustré à la figure A.5. On charge chaque processeur fils de l'arbre de chargement, et l'on attend que tout le sous-arbre correspondant soit chargé pour charger un autre fils. Une fois les sous-arbres chargés, un compte-rendu est envoyé au processeur père.

A.2.3 Phase d'envoi des tables de routage

Le processeur racine exécute le programme illustré à la figure A.6. Il lit l'ensemble des tables de routage et envoie à la suite la table de chacun des processeurs.

```
void load_kernel(kernel)
int kernel;  /* fichier contenant le code du noyau */
{
  uchar kernel_code[KERNEL_SIZE];
  read(kernel, kernel_code, KERNEL_SIZE);
  put_block(KERNEL_SIZE, kernel_code);
  get_block(sizeof(ACK), &ACK); /* synchronisation */
}
```

Figure A.4: Phase de chargement du noyau (racine)

```
void load_kernel(void)
{
  int i;
  for (i = 0; i < MAX_CHILDREN; i++)
  {
    if ((children[numero_prive])[i] == IS_TO_BOOT)
    { /* le processeur correspondant au lien i doit recevoir le noyau */
      charger_noyau(i);
      in(i, &ACK, sizeof(ACK)); /* attendre fin de chargement sous-arbre */
    }
  }
  /* envoi de compte-rendu au pere */
  out(boot_link, ACK, sizeof(ACK)); /* pour permettre la synchronisation */
}
```

Figure A.5: Phase de chargement du noyau (réseau)

```
void send_tables(route_file)
int route_file; /* fichier contenant les tables de routage */

  table routing_tables[NB_PROCESSORS];  /* tables de routage */
  int proc; /* numero de processeur */
  int ACK;  /* message de synchronisation */
  /* lecture de l'ensemble des tables de routage */
  read(route_file, routing_tables, ROUTE_TABLE_SIZE)

  for (proc = 1; proc < NB_PROCESSORS; proc++)
  {
    routing_tables[proc].destination = proc;
    put_block(TABLE_SIZE, &routing_tables[proc]);
  }
```

Figure A.6: Phase envoi de tables de routage (racine)

A.2.4 Phase de lancement du noyau

Il suffit enfin d'envoyer un message à chacun des processeurs du réseau afin d'indiquer au noyau qu'il peut commencer son programme. Nous ne présenterons pas les schémas de programmes correspondants.

A.3 Parallélisation de l'algorithme

Le principe de cette parallélisation est d'éviter les étapes de synchronisation globale.

A.3.1 Phase d'initialisation

L'algorithme présenté procède par un parcours en profondeur d'abord. On peut penser à initialiser en parallèle les différentes branches des processeurs fils. Cependant, le parcours en profondeur effectue aussi la numérotation des nœuds, et un algorithme parallèle demanderait que la numérotation soit obtenue par un autre moyen. Les numéros pourraient être inclus dans la table des processeurs fils, mais la taille du programme d'initialisation est très limitée. Nous ne considérons pas cette parallélisation.

A.3.2 Phase de chargement du noyau

L'algorithme de chargement du noyau est identique à celui de la phase d'initialisation. Nous pouvons donc appliquer la même technique de parallélisation, mais la contrainte de numérotation a disparu.

Le code du noyau peut être chargé en parallèle. Chaque processeur exécute alors le programme illustré à la figure A.7.

Dans un premier temps, une fois que le noyau est chargé, on initialise la table de routage (encore vierge à ce moment là) de façon à pouvoir accéder à la racine et aux processeurs du sous-arbre. Dans un second temps, on mène deux activités en parallèle. La première consiste à continuer le chargement du noyau sur chacun des sous-arbres en parallèle. La seconde envoie un compte-rendu au processeur racine et commence le routage selon la table partielle.

A.3.3 Phase d'envoi des tables de routage

Après le chargement du noyau, chaque processeur signale au processeur racine qu'il est prêt à recevoir sa table de routage. En réponse à chacun de ces messages, il suffit d'envoyer la table correspondante. Une telle table sera acheminée de proche en proche en utilisant les tables partielles ou les tables définitives. Les tables définitives sont un sur-ensemble des tables partielles, de façon à garantir que les messages arrivent à destination pendant cette étape.

Nous ne détaillons pas les schémas de programmes correspondants.

A.3.4 Phase de lancement du noyau

L'étape de chargement du noyau opère en même temps son lancement. Il n'est plus besoin d'étape supplémentaire. Une fois que toutes les tables ont été envoyées, des programmes peuvent être chargés.

```
void load_kernel(void)
{
  message ACK;  /* message de synchronisation */
  int i;
  /* initialiser les entrees utiles de la table de routage */
  route_table[racine] = boot_link;
  /* La phase d'initialisation a permis d'associer un intervalle de
     numeros de processeurs a chaque lien de sortie. Completer la table
     de routage de facon a incorporer cette information (NON DETAILLE)
  */
  PAR   /* en pseudo-parallele */
  {
    1: {  /* continuer le chargement du noyau */
      PAR for (i = 0; i < MAX_CHILDREN; i++)  /* en pseudo parallele */
        if ((children[numero_prive])[i] == IS_TO_BOOT)
        { /* le processeur correspondant au lien i doit recevoir le noyau */
          charger_noyau(i);
        }
    }
    2: {  /* compte-rendu et debut du routage */
      /* envoi de compte-rendu a la racine */
      ACK.dest = racine;  /* envoi direct a la racine car le noyau est
                             deja en place sur le pere, et il sait
                             acheminer les messages vers la racine */
      ACK.data = private_number;
      out(boot_link, ACK, sizeof(ACK));

      /* lancer le noyau avec comme table initiale l'arbre d'initialisation
        (NON DETAILLE)
      */
    }
  }
}
```

Figure A.7: Chargement parallélisé (réseau)

A.4 Bilan

Ce problème simple est représentatif de ceux rencontrés dans la programmation parallèle de bas niveau. Il illustre la complexité de nombreuses situations observées dans le développement de programmes parallèles. Il a donné lieu à une réalisation en langages assembleur et C, à laquelle ont participé M. Favre, D. Fort et Ph. Waille. Le traducteur de langage assembleur a été développé au LGI par J. Eudes.

Remarquons que le nombre de copies a été divisé par N dans la version parallélisée, et correspond alors au nombre minimal de copies requis. D'autre part, la parallélisation de chacune des phases réduit le délai total de chargement.

La phase de chargement du noyau est faite en pipeline avec la phase de chargement des tables de routage. On élimine ainsi le temps nécessaire à une seconde descente de l'arbre de chargement, qui est de l'ordre de N. D'autre part, la version parallèle élimine les deux étapes de synchronisation, qui sont aussi de l'ordre de N. Enfin, le parcours

de l'arbre de chargement est parallélisé, et il passe d'un ordre N à $\log_m N$, m étant le degré de l'arbre de chargement. L'amélioration globale croît avec le nombre de processeurs.

Pour la satisfaction du lecteur intrigué par les choix effectués dans cet algorithme et désireux de l'améliorer, signalons que des travaux ont été menés dans notre équipe [MMS90] dans cette direction par L. Mugwaneza et I. Sakho. Le calcul des tables de routage a été parallélisé et il peut s'exécuter sur le réseau cible. Ainsi, après l'étape de chargement de l'image mémoire, il suffit d'exécuter l'algorithme de calcul des tables.

Sommaire

Liste des Figures

Liste des Tables

Bibliographie

[ABCKSS89] E. A. Arnould, F. J. Bitz, E. C. Cooper, H. T. Kung, R. D. Sansom, P. A. Steenkiste. *The Design of Nectar: A Network Backplane for Heterogeneous Multicomputers.* CMU Report CMU-CS-89-101, April 1989. (cité page 68)

[Accet86] Mike Accetta et al. *MACH: A New Kernel Foundation for UNIX Development.* Computer Science Department, Carnegie Mellon University, 1986. (cité pages 1, 68)

[ACG88] Sudhir Ahuja, Nicholas J. Carriero and David H. Gelernter. *Matching Language and Hardware for Parallel Computation in the Linda Machine.* IEEE Transaction on Computers, Vol. 37, n° 8, Aug. 1988. (cité page 40)

[AGHR89] François Armand, Michel Gien, Frédéric Herrmann and Marc Rozier. *Revolution 89 or "Distributing UNIX Brings it Back to its Original Virtues".* Chorus Systèmes, rapport CS/TR-89-36.1, 1989. (cité pages 1, 68, 137)

[AHLR89] F. Armand, F. Herrmann, J. Lipkis, M. Rozier. *Multi-threaded Processes in Chorus/MIX.* Rapport Chorus Systèmes CS/TR-89-37.3, Oct. 1989. (cité page 70)

[AlGot89] G. S. Almasi and A. Gottlieb. *Highly Parallel Computing.* The Benjamin/Cummings Publishing Company, Inc. 1989. (cité pages 21, 23)

[Andrew81] G. R. Andrew. *Synchronizing Resources.* ACM Trans. Programm. Lang. Syst., 3–4 Oct. 1981, p405–430. (cité page 12)

[ARSha89] Vadim Abrossimov, Marc Rozier and Marc Shapiro. *Generic Virtual Memory Management for Operating System Kernels.* Proceedings of the

12th ACM Symp. on operating Systems Principles, Arizona, 3–6 Dec. 1989. (cité page 85)

[Bach86] Maurice J. Bach. *The Design of the Unix Operating System.* Prentice Hall 1986. (cité pages 61, 62, 99)

[Bal87] R. Balter et al. *Modèle d'exécution du système Guide.* Rapport Guide-R3, Déc. 1987, Bull-LGI-IMAG. (cité pages 5, 61)

[Baron88] Robert V. Baron et al. *MACH Kernel Interface Manual.* Department of Computer Science, Carnegie Mellon University, Feb. 1988. (cité page 67)

[Batcher80] K. Batcher. *Design of a Massively Parallel Processor.* IEEE Transactions on Computers, Vol. 29, n° 9, 1980, p836-840. (cité page 4)

[BBH89] H. Baumgarten, L. Borrmann, H. Hartlage, N. Holt, S. Prior. *Specification of the Process Control Language (PCL).* Rapport EDS.DD.15.0007, Projet ESPRIT EP2025, European Declarative System, 11 Dec. 1989. (cité page 135)

[BDW87] J. Beetem, M. Denneau and D. Weingarten. *The GF11 Parallel Computer.* Experimental Parallel Computing Architectures, North Holland, J. Dongarra Ed. North Holland, Amsterdam 1987. (cité page 24)

[BKT90] H.E. Bal, M.F. Kaashoek and A.S. Tanenbaum. *Experience with Distributed Programming in ORCA.* Proc. International Conf. on Comp. Languages'90, IEEE, 1990. (cité page 61)

[Black86] A. Black et al. *Object structure in the Emerald system.* Proc. of Object-Oriented Programm. Syst. Lang. and Applications, SIGPLAN Not. ACM Vol. 21, n° 11, Nov. 1986, p78–86. (cité pages 12, 68)

[Black90] David L. Black. *Scheduling Support for Concurrency and Parallelism in the Mach Operating System.* IEEE COMPUTER, Vol. 23, n° 6, May 1990, p35–43. (cité page 61)

[Bork89] Shekhar Borkar et al. *iWarp: An Integrated Solution to High-Speed Parallel Computing.* Rapport CMU-CS-89-104, CMU, 11 Janv. 1989. (cité pages 32, 99)

[Botteri91] Thadeu Botteri-Corso. *Un système à la Unix pour multiordinateurs.* Thèse de doctorat, LGI-IMAG, 1991. (à paraître, cité pages 48, 82)

[Braner88] M. Braner. *Trollius Manuals.* Cornel Theory Center, 1988. (cité pages 48, 61, 135)

[Brinch78] P. Brinch Hansen. *Distributed Processes: A concurrent programming concept.* Comm. ACM Vol. 21, n° 11, Nov. 1978, p934–941. (cité page 12)

[BSTan89] H. E. Bal, J. G. Steiner and A. S. Tanenbaum. *Programming Languages for Distributed Computing Systems.* ACM Computing Surveys, Vol. 21, n° 3, Sept. 1989. (cité pages 2, 12, 13, 60)

[CaGel89a] N. Carriero and D. Gelernter. *Linda in Context.* Communications of the ACM, Vol. 32, n° 4, April 1989. (cité pages 18, 61)

[CaGel89b] N. Carriero and D. Gelernter. *How to Write Parallel Programs: A Guide to the Perplexed.* ACM Computing Surveys, Vol. 21, n° 3, Sept. 1989. (cité pages 10, 12)

[Chase89] Jeffrey S. Chase et al. *The Amber System: Parallel Programming on a Network of Multiprocessors.* ACM 1989. (cité page 68)

[Cher88] D. R. Cheriton. *The V Distributed System.* Communications of the ACM, Vol. 31, n° 3, March 1988. (cité page 68)

[Chor87] Chorus Systèmes. *CHORUS Kernel V3.1: Specification and Interface.* Chorus Systèmes, report CS/TN-87-25.3, April 1989. (cité page 79)

[CORNA81] Cornafion (nom collectif). *Systèmes informatiques répartis.* Dunod Informatique, 1981. (cité page 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 pages 33, 93, 95)

[ELS88a] Jan Edler, J. Lipkis, E. Schonberg. *Process Management for Highly Parallel Unix Systems.* Proc.

Workshop on Unix and Supercomputers, Usenix, Pittsburg, 26–27 Sept. 1988, p1–18. (cité pages 28, 69)

[ELS88b] Jan Edler, J. Lipkis, E. Schonberg. *Memory Management in Symunix II: A Design for Large-Scale Shared Memory Multiprocessors.* Proc. Workshop on Unix and Supercomputers, Usenix, Pittsburg, 26–27 Sept. 1988, p151–168. (cité pages 28, 67)

[Eudes90] Jacques Eudes. *PDS : Un Générateur de système de développement pour machines multiprocesseurs.* Thèse de doctorat, LGI-IMAG, 1990. (cité page 81)

[Ferra90] Richard D. Ferrante. *Parameters for Distributed Systems Design.* Journal of Parallel and Distributed Computing 9, 1990, p323–329. (cité page 21)

[Fly72] Michel J. Flynn. *Some Computer Organizations and their Effectiveness.* IEEE Transactions on Computers, Vol. C-21, 1972, p948–960. (cité page 22)

[Frey91] André Freyssinet. *Architecture et réalisation d'un système réparti à objets.* Thèse de doctorat, BULL-IMAG/LGI, Grenoble 1991. (cité pages 12, 68)

[Gar87] N. H. Garnett. *Helios: an Operating System for the Transputer.* Proc. of OUG-7, IOS, Springfield, 1987. (cité pages 48, 61, 135)

[GeKle80] Mario Gerla and Leonard Kleinrock. *Flow Control: A Comparative Survey.* IEEE Transactions on Communications, Vol. COM-28, n° 4, April 1980. (cité pages 93, 100)

[Geler85] David Gelernter. *Generative Communication in Linda.* ACM Transactions on Programming Languages and Systems, Vol. 7, n° 1, January 1985. (cité page 12)

[Gonza91] Néstor González Valenzuela. *Noyau de Système pour les ordinateurs massivement parallèles : contrôle de la communication entre processus.* Thèse de doctorat INPG, LGI-IMAG, 1991. (cité pages 58, 92, 93, 97, 106)

[Ghosal90] Dipak Ghosal and al. *Characterizing Parallel Program Behavior: An Experimental Study.* Institute for Advanced Computer Studies, Computer Science Dept., Univ. of Maryland, College Park, (UMIACS-TR), 1990. (cité page 65)

[HJMW87] G. Harp, C. Jesshope, T.Muntean and C. Whitby-Strevens. *Supernode: development and application of a low cost high performance multiprocessor machine.* Proc. ESPRIT86 Conference, Elsevier, Bruxelles 1987. (cité page 5)

[Hoare78] C.A.R. Hoare. *Communicating Sequential Processes.* Communications of the ACM, Vol. 21, n° 8, Aug. 1978. (cité pages 11, 61)

[Hord82] R. Michael Hord. *The Illiac IV: The first Supercomputer.* Computer Science Press, 1982. (cité pages 3, 24)

[Hudack86] P. Hudack. *Para-functional programing.* Computer 19, 8 Aug. 1986, p60–70. (cité page 12)

[Hwang87] Kai Hwang. *Advanced Parallel Processing with Supercomputers Architectures.* Proceedings of the IEEE, Vol. 75, n° 10, Oct. 1987. (cité page 4)

[Ichbiah79] J. D. Ichbiah and al. *Rationale for the Design of the Ada programming Language.* SIGPLAN Not. 14, 6, June 1979. (cité page 12)

[Inmos84b] David May. *occam Definition.* Inmos Notice, 1984. (cité pages 12, 61)

[Inmos87a] Inmos Ltd. *IMS T800 transputer.* (cité page 88)

[Inmos88] Inmos Ltd. *Transputer Development System.* Prentice Hall, 1988. (cité pages 9, 47)

[IsBor90] Petro Istavrinos and Lothar Borrmann. *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 , Sept. 1990. (cité pages 61, 135)

[Jul88] E. Jul, H. Levy, N. Hutchinson and A. Black. *Fine-Grained Mobility in the Emerald System.* ACM Trans. on Comp. Syst., Vol. 6, n° 1, Feb. 1988, p109–133. (cité page 70)

[KeKle79] Paviz Kermani and Leonard Kleinrock. *Virtual Cut-Through: A New Computer Communication Swithching Technique.* Computer Networks 3, 1979, p267–286. (cité page 95)

[King88] Paul J. King. *Implementing a POSIX Compatible Operating System on a Multi-transputer Supercomputer.* EUUG, 3–7 Oct. 1988, p39–51. (cité pages 48, 61, 135)

[Krako88] Sacha Krakowiak. *Principles of Operating Systems.* The MIT Press, 1988. (cité page 2)

[Kuck78] D. J. Kuck. *The Structure of Computers and Computations.* John Wiley 1978. (cité page 22)

[Langue87] Yves Langué Tsobgny. *Un modèle pour la diffusion basé sur les ensembles de refus.* Rapport de DEA, LGI-IMAG Juillet 1987. (cité page 110)

[Langue89] J. Briat, M. Favre, D. Fort, N. Gonzàlez-Valenzuela, Y. Langué, T. Muntean, Ph Waille. *PARX: A Parallel Operating System for Transputer Based Machines.* Proc. of 10th OUG, 3–5 April, IOS, Springfield, Amsterdam 1989. (cité page 92)

[Langue89] Yves Langué et al. *Un système Unix-like pour une station de travail à base de transputers.* Actes du Séminaire Franco-Brésilien sur les Systèmes Informatiques Répartis, 11–14 Sept. 1989, Florianapolis, Brésil. (cité page 92)

[Lazow81] Edward D. Lazowska et al. *The architecture of the Eden System.* Proc. 8th Symp. on Operating Systems Principles, Dec. 19981, p148–159. (cité page 5, 34, 40)

[Liskov88] Barbara Liskov. *Distributed Programming in Argus.* Communications of the ACM, Vol. 31, n° 3, March 1988. (cité page 12)

[MagMun89] A.C. Magalhaes de Melo Balaniuk, T. Muntean. *Basic Mechanisms for the Supernode File System.* SupernodeI, EP1025, Working Paper 0589, LGI-IMAG, Mai 1989. (cité pages 36)

[MaPu88] H. Massalin, C. Pu. *Fine-Grain Scheduling.* Columbia University TR-CUCS-381-88, Nov. 1988. (cité pages 16, 77)

[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 page 78)

[Menn88] F. Menneteau. *Allocateur de processus sur une architecture parallèle.* Rapport de DEA Informatique, LGI-IMAG, Sep. 1988. (cité page 81)

[MeSea70] R. A. Meyer and L. H. Seawright. *A Virtual Machine Time-Sharing System.* IBM Systems Journal, Vol. 9, n° 3, 1970, p199–218. (cité page 61)

[Milner80] R. Milner. *A Calculus of Communicating Processes.* Lecture Notes in Computer Science 92, Springer Verlag, New York 1980. (cité page 12)

[MMS90] L. Mugwaneza, T. Muntean and I. Sakho. *A deadlock-free routing algorithm with network size independent buffering space.* CONPAR90-VAPPIV, Sept. 1990, Zurich. (cité pages 92, 93,156)

[Mulder88] J.C. Mulder. *On the Amoeba Protocol.* Report CS-R8827 , July 1988 Computer Science Department of Software Technology, Centrum voor Wiskunde en Informatica (CWI), Netherlands. (cité page 94)

[Mullen90] S.J. Mullender et al. *Amoeba: A distributed Operating System for the 1990s.* IEEE COMPUTER, Vol. 23, n° 6, May 1990, p44–53. (cité pages 68, 79)

[Munt88] Traian Muntean. *Les Supercalculateurs à Transputers.* La Recherche n° 204, Vol. 19, Novembre 1988. (cité page 10)

[NBW88] J. R. Nicol, G. S. Blair, J. Walpole. *Operating System Design: Towards a Holistic Approach.* SIGOPS, 1988, p11–19. (cité page 21)

[OAK90] Brian Oakley. *The challenge of the large parallel processors in Europe.* Supercomputer Vol. 7, n° 2, March 1990, p18–28. (cité page 135)

[OSS80] J. K. Ousterhout, D. A. Scelza and P. S. Sindhu. *Medusa: An Experiment in Distributed Operating*

System Structure. Communications of the ACM, Vol. 23, n° 2, Feb. 1980, p92–105. (cité pages 34)

[Pfister86] G. F. Pfister et al. *An Introduction to the IBM Research Parallel Processor Prototype (RP3).* Technical Report, IBM T.J. Watson Research Center Yorktown Heights, NY, 1986. (cité pages 5, 27)

[Pount90] Dick Pountain. *Virtual Channels: The next Generation of Transputers.* BYTE, April 1990. (cité pages 35, 88)

[Pritch87] David J. Pritchard. *Mathematical Models of Distributed Computation.* OUG-7, IOS, Amsterdam Springfield, 14–16 Sept. 1987. (cité page 10)

[ReFu87] D. A. Reed and R. M. Fujimoto. *Multicomputer Networks: Message-Based Parallel Processing.* The MIT Press, 1987. (cité pages 21, 34, 37)

[RiSh90] J. Richard Sherman. *Recognising the Challenges of Change.* Supplement Proceedings, Second European Exhibition and Conference on Supercomputing, Supercomputing Europe'90, 10–12 Jan. 1990, p56–82. (cité page 135)

[Riveill87] Michel Riveill. *CONKER : un modèle de répartition pour processus communicants, Application à occam.* thèse de Doctorat INPG, 13 Fév. 1987, Grenoble. (cité page 18)

[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 pages 36, 39)

[SaSchu88] Youcef Saad and Martin H. Schultz. *Topological Properties of Hypercubes.* IEEE Transactions on Computers, Vol. 37, n° 7, July 1988, p867–872. (cité page 31)

[Schon87] Willi Schonauer. *Scientific Computing on Vector Computers* Special Topics in Supercomputing, Vol. 2, Elsevier, North Holland, Amsterdam 1987. (cité page 21)

[Shapiro86] E. Shapiro. *Concurrent Prolog: A progress report.* Computer 19, 8 Aug. 1986, p44–58. (cité pages 12, 19)

[Shea89] D. Shea et al. *The IBM Victor Multiprocessor Project.* Proc. of the 4th International Conference on Hypercubes, April 1989. (cité page 25)

[Shizgal87] I. Shizgal. *An Amoeba Replicated Service Organization.* CWI, Report CS-R8723, May 1987, Netherlands. (cité page 107)

[SnI89] LGI-IMAG. *Allocator & Mapper for the Supernode Program Development System.* Deliverable Report Supernode I EP1085, LGI-IMAG, Jan. 89. (cité pages 5, 81)

[StYem83] R. E. Strom, S. Yemini. *NIL: An Integrated language and system for distributed programming.* SIGPLAN Not. ACM Vol 18, n° 9, June 1983, p73–82. (cité page 12)

[SZH85] D. C. Swinehart, P. T. Zellweger, R. B. Hagmann. *The structure of Cedar.* Proc. ACM SIGPLAN 85, Symp. on Language Issues in Programming Environments, SIGPLAN Not. ACM Vol. 20, n° 7, July 1985, p230–244. (cité page 12)

[TaMun90] E-G. Talbi, T. Muntean. *Placement statique de processus sur une architecture parallèle.* Rapport de recherche RR 833-I-, LGI-IMAG, INPG, Nov. 1990. (cité page 84)

[TaMun91] E-G. Talbi, T. Muntean. *Un algorithme d'allocation dynamique de processus sur une architecture parallèle à mémoire distribuée.* Rapport de recherche (à paraître), LGI-IMAG, INPG, Mars 1991. (cité page 84)

[Tanen87] A. S. Tanenbaum. *Operating Systems: Design and Implementation.* Prentice Hall, 1987. (cité pages 48, 62, 83)

[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 pages 67, 79)

[Teva87b] 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 page 70)

[Tevan87] Avadis Tevanian Jr. *Architecture-Independant Virtual Memory Management for Parallel and Distributed Environments: The Mach Approach.* Doctor of Philosophy thesis, CMU, Déc. 1987. (cité page 67)

[TBH85] P. C. Treleaven, D. R. Brownbridge and R. P. Hopkins. *Data Driven and Demand Driven Computer Architecture.* ACM Computing Surveys, Vol. 14, n° 1, Marsh 1982, p95–143. (cité page 22)

[TuRo88] Lewis W. Tucker and Georges G. Robertson. *Architecture and Applications of the Connection Machine.* IEEE COMPUTER, August 1988. p26–38. (cité pages 4, 23)

[Valliant82] L. G. Valliant. *A Scheme for Fast Parallel Communication.* ACM J. Comput., Vol 11, n° 2, May 1982, p350–361. (cité page 54)

[Waille90] Ph. Waille. *Introduction à L'architecture des Machines Supernode.* Rapport Technique n° 56, LGI-IMAG, Février 1990. (cité page 27, 45)

[Waille91] Ph. Waille. *Architectures Parallèles à Connectique Programmable : Reconfiguration et Routage.* Thèse de doctorat de l'INPG, spécialité Micoélectronique, IMAG-LGI, 11 Sept. 1991. (cité page 58)

[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 pages 34, 61, 91, 135)

[YoTo86] Y. Yokote, M. Tokoro. *The Design and Implementation of ConcurrentSmalltalk.* Proc. of Object-Oriented Programm. Syst. Lang. and Applications, SIGPLAN Not. ACM Vol. 21, n° 11, Nov. 1986, p331–340. (cité page 12)

[Young87] Michael Young et al. *The Duality of Memory and Communication in the Implementation of a Multiprocessor Operating System.* ACM SIGOPS, 1987. (cité page 22)

Grenoble, le 3 Décembre 1991

ECOLE DOCTORALE

Affaire suivie par : Michèle SIMEON
Tél : 76.57. 4525
JPV/MS

N/Réf. :

Objet :

AUTORISATION de SOUTENANCE

Vu les dispositions de l'arrêté du 23 Novembre 1988 relatif aux Etudes Doctorales
Vu les rapports de présentation de :

- Mr André SCHIPER Professeur Ecole Polytechnique Fédérale de LAUSANNE
- Mr Claude BETOURNE Université P.Sabatier Laboratoire LSI TOULOUSE

Mr LANGUETSOBGNY Yves

est autorisé(e) à présenter une thèse en soutenance en vue de l'obtention du diplôme de DOCTEUR de l'INSTITUT NATIONAL POLYTECHNIQUE de GRENOBLE, spécialité :

"Informatique"

Pour le Président de l'I.N.P.-G.
et par délégation,
le Vice-Président
M. GARNIER