THESE

présentée par

MACIEL DA COSTA Celso

pour obtenir le titre de

DOCTEUR de l'UNIVERSITE JOSEPH FOURIER GRENOBLE I

(Arrêté ministériel du 30 Mars 1992)

Spécialité : INFORMATIQUE

ENVIRONNEMENT D'EXÉCUTION PARALLÈLE : CONCEPTION ET ARCHITECTURE

Date de soutenance : 21 Octobre 1993

Composition du jury :

Président	Jacques	MOSSIÈRE
Rapporteurs	Françoise	ANDRÉ
	Claude	BETOURNÉ
Examinateurs	Brigitte	PLATEAU
	Jacques	BRIAT

Thèse préparée au sein du Laboratoire de Génie Informatique

Je tiens à remercier :

- Monsieur Jacques Mossière, Professeur à l'Institut National Polytechnique de Grenoble, qui m'a fait l'honneur de présider mon jury de soutenance,

- Monsieur Jacques Briat, Maître de Conférence à l'Université Joseph Fourier, sans qui cette thèse n'aurait pas pu voir le jour. Je veux lui exprimer ma profonde gratitude et reconnaissance pour son accueil et son soutien au sein de son équipe, pour les idées qu'il m'a suggéré tout au long de mon travail de recherche, et pour ses conseils lors de la rédaction de ce document,

- Monsieur Claude Betourné, Professeur à l'Université Paul Sabatier de Toulouse, et Madame Françoise André, Professeur à l'Université de Rennes, pour avoir accepté d'être rapporteurs de cette thèse, et par leur lecture attentif de mon document,

- Madame Brigitte Plateau, Professeur à l'Institut National Polytechnique de Grenoble, pour l'intérêt qu'elle a porté à mon travail en acceptant de participer à ce jury,

- Salah Edinne Kannat, Eric Morel, Alexandre da Silva Carissimi et Michel Favre pour leur aide et soutien dans mon travail de recherche,

- à tous mes collègues du Laboratoire de Génie Informatique du site Viallet.

A ma femme et mes enfants.

Résumé

L'objectif de cette thèse est l'étude d'un environnement d'exécution pour machines parallèles sans mémoire commune. Elle comprend la définition d'un modèle de programme parallèle, basé sur l'échange de message offrant une forme restreinte de mémoire partagée. La communication est indirecte, via des portes; les processus utilisent les barrières pour la synchronisation. Les entités du système, processus, portes et barrières, sont créées dynamiquement, et placées sur un processeur quelconque du réseau de processeurs de façon explicite.

Nous proposons une implantation de ce modèle comme la mise en oeuvre systématique d'une architecture client/serveur. Cette implantation a été effectuée sur une machine Supernode. La base est un Micro Noyau Parallèle, où le composant principal est un mécanisme d'appel de procédure à distance minimal.

Mots clefs : Machines parallèles à mémoire distribuée, Modèles de programmes parallèles, Environnement d'exécution, Modèle client/serveur, Appel de procédure à distance, Micro Noyau Parallèle.

Abstract

This thesis describes an execution environment for parallel machines without shared memory. A parallel programming model based on message passing, with a special shared memory. In this model, process communication occurs indirectly, via ports, and the processes use barriers for synchronization. All the entities of the system, such as processes, ports and barriers, are created dynamically and loaded on any processor of the network of processors.

The implementation architecture of our model is a systematic realization of the client/server model. An implementation is proposed in a Supernode parallel machine as a parallel micro kernel. The principal parallel micro kernel component is a minimal remote procedure call mechanism.

Key words : Distributed memory parallel machine, Parallel programming model, Execution environment, Client/server model, Remote procedure call, Parallel Micro Kernel.

Table des matières

1 Introduction **9**

2 Les environnements de programmation parallèle **13**

2.1 Introduction 13

2.2 Modèles de machines parallèles 13

2.3 Les modèles de programmation parallèle 16

2.3.1 Expression du parallélisme 17

2.3.2 La communication 19

2.3.3 La synchronisation 22

2.4 Les environnements de programmation 25

2.5 Le langage Concurrent C 27

2.5.1 Le modèle de programme 27

2.5.2 Expression du parallélisme 28

2.5.3 La communication et la synchronisation 29

2.5.4 L'environnement de programmation 34

2.5.5 L'environnement d'exécution 34

2.6 Le langage Orca 37

2.6.1 Le modèle de programme 38

2.6.2 Expression du parallélisme 38

2.6.3 La communication et la synchronisation 39

2.6.4 L'environnement de programmation 43

2.6.5 L'environnement d'exécution 44

2.7 Le langage Linda 46

2.7.1 Le modèle de programme 46

2.7.2 Expression du parallélisme 46

2.7.3 La communication et la synchronisation 47

2.7.4 L'environnement de programmation 51
2.7.5 L'environnement d'exécution 52
2.8 p4 Programming System 54
2.8.1 Le modèle de programme 54
2.8.2 Expression du parallélisme 54
2.8.3 La communication et la synchronisation 55
2.8.4 L'environnement de programmation 57
2.8.5 L'environnement d'exécution 57
2.9 Conclusion 58

3 Un modèle de programme parallèle **63**
3.1 Introduction 63
3.2 Expression du parallélisme 64
3.2.1 Tâches 65
3.2.2 Processus 67
3.2.3 Contrôle de l'exécution parallèle 68
3.3 La communication 70
3.3.1 Les opérations sur les portes 71
3.4 La synchronisation 72
3.5 Les actions pluralistes 73
3.5.1 Les opérations sur les listes 74
3.5.2 Réplication de tâches 76
3.5.3 Réplication de processus 76
3.5.4 Diffusion d'un message 77
3.5.5 Écriture multiple 77
3.5.6 Attente multiple 77
3.6 L'organisation des programmes parallèles 78
3.6.1 Parallélisme de contrôle 78
3.6.2 Parallélisme de données 80
3.6.3 Processus communicants 83
3.6.4 Maître/travailleurs 84
3.7 Comparaison avec les systèmes étudiés 85
3.7.1 Le protocole RPC 85
3.7.2 Espace de tuples 86
3.7.3 Bilan 87

4 L'environnement d'exécution **89**
4.1 Introduction 89
4.1.1 Présentation de la machine parallèle 89
4.1.2 Concepts et architecture 90
1 Micro Noyau Parallèle 90
2 Micro Noyau de Communication 92
4.2 Le Micro Noyau Parallèle 100
4.2.1 Le Micro Noyau local 100
4.2.2 Le Micro Noyau Parallèle 102
L'adressage des serveurs 103
Les paramètres 103
La définition d'un service chez le client 104
La définition d'un service chez le serveur 106
Le message 108
4.2.3 Le micro noyau de communication 109
L'organisation du noyau de communication 109
1 La couche protocole 111
2 La couche acheminement de messages 115
4.2.4 Bilan 119
4.3 Les services 119
4.3.1 Organisation générale des services 120
4.3.2 Interaction entre serveurs 121
4.3.3 Gestion des entités 122
4.3.4 Le service de synchronisation 123
Les procédures de service 124
4.3.5 Le service de gestion des tâches et processus 125
Graphe d'état des processus et tâches 125
Descripteur d'une tâche/processus 126
Les procédures de service 127
4.3.6 Le service de communication 130
Les portes Data 130
Les portes Message 132
4.3.7 Le service de diffusion 135
4.3.8 Bilan 136

5 L'environnement de programmation **139**
5.1 Introduction 139
5.2 L'environnement matériel et la configuration initiale 140
5.3 Le chargement 141
5.4 L'accès aux services externes 143
5.5 Conclusion 144

6 Conclusion **145**
6.1 Bilan 145
6.2 Perspectives 147

Bibliographie **149**

Liste des Figures

4.1 La machine Tnode à 16 Transputers. 90
4.2 Organisation typique d'un noyau monolithique parallèle. 91
4.3 Structure d'un Micro Noyau. 91
4.4 La communication entre client et serveur distant dans un Micro Noyau Parallèle. 92
4.5 La couche protocole. 93
4.6 Echange de messages entre client et serveur local. 95
4.7 Echange de messages entre client et serveur distant. 95
4.8 Format d'un message d'APPEL. 108
4.9 Format d'un message de REPONSE. 109
4.10 Noyau de communication avec un seul émetteur. 110
4.11 Noyau de communication avec plusieurs émetteurs. 111
4.12 Les opérateurs de synchronisation. 124
4.13 Les opérations de gestion de tâches et processus. 125
4.14 Graphe d'état des processus. 125
4.15 Graphe d'état des tâches. 126
4.16 Les opérations sur les portes Data. 131
4.17 Graphe d'état des portes Data. 131
4.18 Les opérateurs de communication. 132

5.1 Organisation générale de l'environnement matériel. 140
5.2 La machine Supernode configurée en anneau. 141

Liste des Tables

2.1 Puissance relative des gardes de Concurrent C et Orca. 43
2.2 Types d'appels des services. 43
2.3 Création dynamique des entités. 43
2.4 Communication et synchronisation. 60
2.5 Expression du parallélisme, placement et migration. 60
2.6 Modèles de programmes et machines cible. 61

3.1 Les principaux aspects des systèmes étudiés. 88

Chapitre 1

Introduction

S'il est vrai que le parallélisme est un moyen d'accroître la puissance de calcul, il ajoute d'importantes difficultés à la programmation. Concevoir et mettre au point des programmes parallèles est une tâche très difficile. Le programmeur doit arriver à élaborer un algorithme parallèle efficace qui tire avantage de la puissance de calcul d'une machine parallèle. Pour exprimer son algorithme il a besoin d'un modèle de programme parallèle caracterisé par la façon de faire coopérer les processus d'un algorithme parallèle. Celui-ci s'exprime par un langage de programmation. Les langages utilisés aujourd'hui, dans leur presque totalité, ont été développées à partir de langages sequentiels et sont influencés par les différents types de machines cibles. Des nouveaux langages, crées spécialement pour traiter le parallèlisme sont souhaitables.

Une autre source de problème provient des lacunes des environnements de programmation. Les utilisateurs ont besoin d'outils d'édition et visualisation, de preuve et mise au point, de prise de mesures et d'évaluation, et de placement qui permettent le développement interactif de programmes corrects et éfficaces. Seule l'intégration de tous ces outils permettra d'obtenir un véritable environnement de programmation parallèle. Le développement d'un tel environnement est à l'heure actuelle un important sujet de recherche, car sa réalisation conditionne la généralisation de l'usage du parallélisme.

En ce qui concerne les systèmes d'exploitation, pour les machines à mémoire commune il est possible de récupérer une partie des systèmes et des outils de

programmation classique. Par contre, pour les machines à mémoire distribuée, de nouveaux systèmes doivent être conçus. La construction d'un tel système consiste à traiter tous les problèmes concernant la gestion des processus parallèles, la communication entre processus et la synchronisation, dans le cadre d'un réseau de processeurs.

Parmi les machines sans mémoire commune, les machines parallèles à base de Transputers offrent un rapport qualité/prix intéressant. Les problèmes précédents sont encore accentués par le caractère atypique du Transputer, et l'absence de système et logiciels classiques.

Depuis plusieurs années l'équipe PLoSys (Parallel Logic System) du Laboratoire de Génie Informatique s'intéresse à ces problèmes. Le groupe a participé à la conception de la machine parallèle Supernode dans le cadre du projet ESPRIT 1085. Elle a débuté dans ce cadre par la conception et la réalisation d'un environnement d'exécution parallèle pour Prolog. La maquette OPERA (OU parallélisme et régulation adaptative) [Bri90a], [Bri90b], [Bri91], [Gey91], [Fav92] exploitait le parallélisme Prolog sans intervention du programmeur. Pour cette implantation le groupe a du réaliser un ensemble d'outils de base indispensables au développement de Prolog :

- un assembleur pour le Transputer (processeur de la machine Supernode),
- un chargeur/éditeur de liens,
- un compilateur C pour le Transputer, avec des extensions pour la programmation concurrente,
- une bibliothèque de fonctions C,
- un noyau d'exécution minimum de Prolog.

Du fait de l'expérience acquise dans le développement de cet environnement, le groupe PloSys a décidé d'étudier la conception d'un environnement parallèle pour langage impératif classique (C par exemple). Notre travail de thèse est l'étude du noyau exécutif d'un tel environnement de programmation parallèle, c'est à dire, l'ensemble des opérateurs permettant d'étendre un langage C en un langage parallèle. L'objectif de notre travail comportait deux volets :

- définir un environnement d'exécution parallèle,

- proposer et utiliser une méthode de construction systématique de cet environnement à partir de mécanismes très élémentaires.

Cet **environnement d'exécution parallèle** repose sur un **modèle de programme parallèle**. Le modèle que nous avons conçu est basé sur **l'échange de messages**. Il offre aussi une forme restreinte de **mémoire partagée**. La **communication** est **indirecte** via les **portes**. La **mémoire partagée** est constituée par des portes particulières. La synchronisation est assurée par des **barrières**. La **communication et la synchronisation** peuvent concerner plusieurs processus via une forme restreinte du concept de **groupe**. Les entités **processus**, **portes** et **barrières** peuvent être créés **dynamiquement** sur n'importe quel site. Le placement est explicite et ne peut pas être remis en cause.

L'environnement d'exécution implantant ce modèle a été construit à l'aide d'un Micro Noyau Parallèle. Nous avons ensuite étudié l'architecture d'une implantation de ce modèle comme une mise en oeuvre systématique d'un modèle client-serveur. Nous avons proposé une implantation de celui-ci pour la machine Supernode, sur un Micro Noyau Parallèle, dont le composant principal est un mécanisme élémentaire d'appel de procédure à distance.

Le plan de cette thèse est le suivant :

- Le chapitre deux est dédié à l'étude de quelques environnements de programmation parallèle : Concurrent C, Orca, Linda et du système p4. Ce choix n'est pas exhaustif. Ces environnements ont été choisis pour leurs caractéristiques très différentes. On y détaille les aspects les plus importants pour l'expression du parallélisme, pour la communication et pour la synchronisation.

- Le chapitre trois présente notre modèle de programme pour machines NORMA. On y propose un jeu complet d'opérateurs spécifiques que nous pensons être un compromis intéressant entre les différentes possibilités décrites au chapitre précédent.

- Le chapitre quatre décrit l'architecture de notre implantation sur une machine à mémoire distribuée. Il se décompose en deux parties : la première décrit notre Micro Noyau parallèle. La seconde partie décrit l'utilisation des différents serveurs implantant notre modèle.

- Le chapitre cinq est consacré à une brève présentation de l'environnement d'exploitation de la machine parallèle, qui permet d'y accéder, de charger le Micro Noyau Parallèle et de lancer des applications parallèles.

- Finalement, le chapitre six est destiné à la conclusion. Nous y faisons un bilan de notre étude et nous présentons quelques possibilités de poursuite des travaux.

Chapitre 2

Les environnements de programmation parallèle

2.1 Introduction

Dans ce chapitre nous présentons des caractéristiques principales des **modèles de machines parallèles**, ainsi que les **modèles de programmation** existants, qui sont déterminés par ces caractéristiques. Nous présentons aussi une classification des **environnements de programmation parallèle**.

Nous décrirons ensuite quelques systèmes existants, du point de vue solutions adoptées face aux aspects critiques identifiés. Cette étude nous fournit les bases pour la définition et l'implantation de notre environnement de programmation parallèle.

2.2 Modèles de machines parallèles

Flynn[Fly72] a classé les architectures de machines parallèles en deux types :

- SIMD
- MIMD

Les ordinateurs SIMD (Single Instruction, Multiple Data) sont des machines séquentielles où chaque étape est l'exécution par un grand nombre de processeurs de

la même instruction sur des données particulières. Ce type de machine possède une structure de contrôle séquentielle avec des opérateurs qui fonctionnent en parallèle. On peut construire des environnements pour faire de la parallélisation automatique de programmes séquentiels sur ces machines. Les programmes les plus adaptés à cette parallélisation sont ceux ayant une structure répétitive sur des données de structure régulière. C'est généralement un parallélisme à grain fin. Les machines SIMD sont les machines parallèles les plus anciennes et leur utilisation a fait l'objet d'un travail important sur les modèles, les langages de programmation et les algorithmes adaptés à ces machines.

Les machines MIMD (Multiples Instructions, Multiples Data) sont composées de processeurs qui opèrent de manière indépendante. Il n'y a plus de contrôle unique et un grand nombre de formes de parallélisme est possible. Classiquement, on distingue les machines MIMD à mémoire commune et celles à mémoire distribuée. Le grand problème des machines à mémoire commune est le goulot d'étranglement que constitue l'accès à cette mémoire commune. On connaît deux types d'architecture de machines à mémoire commune : UMA et NUMA. Les machines UMA (Uniform Memory Architecture[Hag92]) préservent un temps uniforme d'accès à la mémoire et ceci impose une forte contrainte sur l'architecture du médium d'accès à la mémoire. Dans les machines NUMA (Non Uniform Memory Architecture) la mémoire est distribuée sur l'ensemble des processeurs. Le temps d'accès à la mémoire est non uniforme, c'est à dire, les accès sont plus ou moins rapides selon que la mémoire adressée est locale ou non au processeur. Les machines UMA/NUMA :

- utilisent des processeurs standards,
- disposent d'une mémoire hiérarchisée d'un à plusieurs niveaux de caches,
- possèdent un dernier niveau commun de mémoire,
- offrent un grain fin de partage de données (l'accès au mot) et un grain moyen de parallélisme (commutation de processus légers),
- ont un degré de parallélisme limité par le goulot d'accès à la mémoire et par la contrainte du maintien de la cohérence de caches.

Dans les machines MIMD sans mémoire commune (NORMA - NO Remote Memory Architecture) chaque processeur possède sa propre mémoire et ne peut pas accéder à la mémoire d'un autre processeur. Ces machines sont composées par des processeurs standards et offrent un grain gros de parallélisme. La notion de machine NORMA est générique. Elle inclut aussi bien un ensemble de stations de travail connectées par un réseau local, qu'une machine composée de microprocesseurs interconnectés, par exemple, en hypercube.

Les machines du type UMA/NUMA permettent une programmation parallèle confortable, car il est possible de récupérer les systèmes et logiciels des machines traditionnelles. Leur seul grand problème est celui de l'**ordonnancement optimal** des processus d'un programme parallèle.

Par contre, préalablement à la résolution de ce problème, les machines NORMA nécessitent une programmation spécifique et la réalisation d'un système distribué sur le réseau de processeurs. Les problèmes à résoudre de façon préliminaire concernent la **gestion des processus parallèles**, la **communication entre ces processus**, et la **synchronisation**. De plus, des **environnements de programmation** parallèle sont nécessaires pour la maîtrise de ce type de programmation. Un problème important de la **gestion des processus parallèles** est le **placement** des processus sur les processeurs. Ce **placement** peut être **statique** ou **dynamique**. Le placement dynamique peut se faire au démarrage d'un processus ou au cours de sa vie. Dans ce cas, un mécanisme de **migration** est nécessaire. Un choix de **placement** suppose par ailleurs une politique de **régulation de charge**.

En ce qui concerne la **communication**, les problèmes portent sur le **routage** et de l'**engorgement** des communications dans un réseau, la **prévention de l'interblocage**, mais aussi les **protocoles de communication** entre processus (**bi-point, diffusion, etc.**).

Les problèmes de la **synchronisation** portent sur le maintien de propriétés portant sur des informations **distribuées**. Cette vérification doit se faire efficacement, c'est à dire, en minimisant les communications. Ceci se traduit par des contraintes supplémentaires sur les communications (ex. préservation de l'ordre causal).

Outre des **langages de programmation** et **compilateurs**, un **environnement de programmation** doit offrir des outils pour la **vérification**, pour la **prédiction de performances**, pour la **mise en oeuvre** et pour **l'exploitation de la machine**.

A l'heure actuelle, ces difficultés sont encore un obstacle à l'utilisation massive des machines parallèles NORMA.

2.3 Les modèles de programmation parallèle

Pour exprimer un algorithme parallèle on a besoin d'un **modèle de programme**. Le type de machine (UMA/NUMA, NORMA) a une influence forte sur les modèles de programmation existants. En effet, pour compiler, il faut être capable de traduire le parallélisme exprimé dans le programme dans le parallélisme de la machine cible. La recherche privilégiée de la facilité de compilation imposera donc des modèles voisins de l'architecture cible.

Dans le **modèle de programme pour les machines à mémoire partagée** les processus partagent la mémoire. Ils coopèrent en utilisant des données partagées et la synchronisation est réalisée par les concepts et opérateurs traditionnels de sémaphores et moniteurs.

Comme les machines UMA/NUMA sont bien adaptées au modèle de programmation à **mémoire partagée**, elles sont faciles à programmer. La seule limitation au parallélisme est le ralentissement induit par une surcharge des voies d'accès à la mémoire partagée.

Du fait de l'absence de cette mémoire commune, la coopération des processus dans une machine à mémoire distribuée se fait par **l'échange de messages**. Si un processus veut envoyer une structure de données à un autre processus qui s'exécute sur un autre processeur, il doit regrouper les données à communiquer en une zone de mémoire contiguë (emballage) et émettre ce bloc. C'est l'opération d'émission. Réciproquement, les opérations du type **receive** permettent à un processus de recevoir un message d'un autre processus c'est à dire, de recevoir un bloc et d'en extraire les données nécessaires (déballage).

Le modèle à **processus communicants** est bien adapté aux machines NORMA. Comme celles-ci offrent un degré de parallélisme plus élevé mais à plus gros grain, le contrôle du grain des processus et leur placement seront les deux grands problèmes d'implantation de ce modèle sur ces machines.

Si le modèle de **processus communicants** est implantable sans problème sur une architecture UMA/NUMA, il n'en est pas de même du modèle à **mémoire partagée** sur une architecture NORMA. En effet cette implantation exige de simuler une mémoire commune sur une architecture composée par un réseau de processeurs. Classiquement cette mémoire est construite soit par une zone accédée à distance soit par des copies multiples dont on doit assurer la cohérence. Ceci est un problème important de l'architecture des systèmes distribués. Sa solution passe par le contrôle du compromis **migration/duplication de données**. Dans ce dernier cas on doit faire face au problème classique dit "**cohérence de copies multiples**".

Un **modèle de programmation parallèle** se manifeste soit par un nouveau langage, soit par des extensions ajoutées à un langage classique ou un langage classique enrichi par l'utilisation de bibliothèques parallèles. Les points clés d'un tel modèle sont l'**expression du parallélisme**, de la **communication** et de la **synchronisation**.

Dans ce qui suit, nous allons répertorier rapidement les différents concepts et opérateurs caractérisant de tels modèles, ainsi que les incidences de l'architecture sur ces concepts et opérateurs.

2.3.1 Expression du parallélisme

Programmer est la traduction d'un algorithme générique et abstrait en un programme destiné à être exécuté par une machine spécifique. Un modèle de programme est un modèle abstrait de la façon de programmer une machine, ou une classe de machines. C'est une abstraction plus ou moins commode de cette machine et de ses traits. Il se présente alors deux choix d'expression du le parallélisme d'un programme : **explicitement** ou **implicitement**.

Avec le **parallélisme explicite**, le programmeur est responsable de l'expression du parallélisme dans son programme (définition et création de processus). La **paral-**

lélisation implicite, par contre, est faite par le compilateur et consiste, à partir d'un programme séquentiel, ou applicatif (fonctionnel, logique ou relationnel), à produire le code parallèle explicite correspondant.

Les arguments en faveur de la parallélisation automatique sont la possibilité de récupérer les programmes existants et les programmeurs formés à la programmation séquentielle. Les utilisateurs continuent à écrire leurs programmes de manière habituelle alors que l'usage des constructions parallèles ajoutent un degré de difficulté à la production et la correction des programmes.

Cependant cette parallélisation automatique pose de très gros problèmes. Il s'agit d'analyser complètement un programme séquentiel afin de trouver les parties du programme pouvant s'exécuter en parallèle et produisant un accroissement de performance par rapport à l'exécution séquentielle. Ainsi, les problèmes de parallélisation automatique de Fortran traitent principalement de la parallélisation de boucles. Par ailleurs, le parallélisme implicite ne fait cependant que déplacer le problème de l'expression du parallélisme. Cette responsabilité est retirée au programmeur et transférée au compilateur par le biais de règles de transformation d'expression séquentielle en expression parallèle. C'est à dire, de règles de transformation d'un modèle séquentiel en un modèle à parallélisme explicite.

Il y a plusieurs façons de définir l'exécution parallèle d'un programme, c'est à dire, l'ensemble des processus qui l'exécutent. Cet ensemble peut être défini **statiquement** et ne pas évoluer au cours de l'exécution du programme. Les processus sont déclarés explicitement, généralement comme une procédure à exécuter ou comme une incarnation d'un modèle de processus, lui même défini comme une procédure. L'intérêt d'un programme comportant un nombre fixe de processus est de permettre le **placement** a priori des processus sur les processeurs. Sinon, cet ensemble initial de processus évolue par **création dynamique** de nouveaux processus et ceux-ci doivent être placés **dynamiquement**.

Une commande classique de création de processus est l'opération **fork**. Le processus qui exécute **fork** (le père) et le processus créé (le fils) continuent en parallèle. Pour se synchroniser avec ses fils, le père exécute une commande **join** d'attente de fin de ses fils. Ce modèle est **asynchrone** (le père continue en parallèle avec le(s) fils) et permet la création **dynamique** d'un nombre quelconque de processus par un père qui choisit d'effectuer la synchronisation de terminaison à son gré.

La commande **cobegin** est la réalisation d'un modèle **synchrone** où le père attend la fin de tous ses fils. Par exemple, la commande

```
cobegin
     commande1 ;
     commande2 ;
     ...
     commanden ;
coend
```

lance l'exécution concurrente des expressions **commande1**, **commande2**, ... , **commanden**. Chaque expression peut contenir des instructions quelconques, y compris d'autres blocs **cobegin/coend**. Le processus qui exécute **cobegin/coend** est suspendu jusqu'à ce que tous ses processus fils terminent (**synchrone**). Le nombre de processus généralement est défini à priori (**statique**).

On distingue actuellement deux niveaux de processus dans l'**expression du parallélisme** : **processus lourd** ou **tâche** et **processus léger** ou **processus**. Un processus lourd est une image mémoire partagée par plusieurs processus légers. Les processus légers sont des processus autonomes, qui ont leur propre pile et leurs propres registres et qui partagent des zones de mémoire (constantes ou variables). Cet ensemble de processus constitue un processus lourd. Les processus lourds ne partagent pas de la mémoire.

Un ensemble de processus lourds communicants peut être vu comme une abstraction de machine NORMA. Un processus lourd supportant plusieurs processus légers est lui même une abstraction de machine virtuelle UMA. Le couple **processus lourd/processus léger** permet donc de prendre en compte une machine dont l'architecture serait hétérogène comme par exemple un réseau de stations multiprocesseurs UMA.

2.3.2 La communication

Un autre aspect important à considérer dans le modèle de **processus communicants** est la façon d'assurer la communication entre processus. Une communication est l'envoi et la réception d'un message. De nombreuses façons de procéder sont possibles. Par exemple, dans le cas où le destinataire n'écoute pas, le message peut être **perdu ou enregistré**. Du point de vue de l'**émetteur**, il peut continuer après son envoi ou être suspendu en attente d'une réponse. En plus, la communication peut

être **directe**, c'est à dire, un message est transmis à un processus ou **indirecte**. Si la communication est **directe**, un message est envoyé à destination d'un processus (ou d'un groupe). Dans la **communication indirecte**, le message est envoyé à une entité intermédiaire : une **boite aux lettres** ou **porte**. Une **porte** peut alors être associée statiquement à un processus (le récepteur) ou avoir un status propre et autonome.

Si la communication exige d'établir un **support spécifique** on parle de **communication** en mode **connecté** et du concept de **canal de communication**. Une connexion doit être établie entre les processus. Si la connexion est établie à l'initiative d'un seul processus, on parlera de connexion **asymétrique**. Si elle peut être établie à l'initiative de tout processus, on parlera de connexion **symétrique**.

Dans une communication **bi-point synchrone**, l'émetteur est bloqué jusqu'à ce que le récepteur reçoive le message. Les processus non seulement **communiquent** mais aussi **synchronisent** leurs actions. Si la communication est **asynchrone**, l'émetteur continue son exécution après l'envoi. Un tampon doit donc stocker les messages qui n'ont pas encore été consommés. Il peut cependant avoir un risque de saturation du tampon par un trop grand nombre de messages non consommés. Ceci implique soit de perdre des messages ou de bloquer l'émetteur jusqu'à obtention d'une place (contrôle de flux). Dans la communication asynchrone il faut décider si l'on préserve l'ordre d'envoi (canal FIFO) ou non.

Lorsque deux processus seulement sont mis en cause dans une communication, on parle de **communication bi-point**. Lorsque tous les processus sont concernés, on parle de communications globales, dont la plus connue est la **diffusion**. Si une partie seulement des processus est concerné, on parle de communication de **groupe**. La **diffusion** est alors restreinte aux membres du groupe. On distingue les communications **internes** et **externes** aux groupes. Une communication **interne** au groupe ne concerne que les processus du groupe. L'organisation d'un groupe est **hiérarchique** s'il y a un représentant du groupe qui reçoit les communications **externes**. Le processus **représentant** du groupe constitue un goulot d'étranglement potentiel ainsi qu'un point critique pour la fiabilité de l'ensemble. Dans un groupe **non hiérarchique**, les processus externes peuvent communiquer directement avec des membres du groupe. Le processus **représentant** du groupe constitue un goulot

d'étranglement potentiel ainsi qu'un point critique pour la fiabilité de l'ensemble. Il y a plusieurs façons de **diffuser** dans un groupe de processus :

1. **Diffusion à la source**, c'est à dire, par autant d'émissions qu'il y a de processus destinataires.

2. **Diffusion progressive**, à partir d'un noyau initial selon un arbre couvrant les processus concernés.

Une optimisation est de restreindre la diffusion aux processeurs supportant les processus du groupe. Chaque noyau diffuse donc localement aux processus. La seconde solution peut exploiter une **diffusion matérielle** ou une géométrie particulière du réseau d'interconnexion. Une autre caractéristique importante de la diffusion est **l'atomicité**. La **diffusion est atomique** si tous les membres du groupe reçoivent une série de diffusions dans le **même ordre**. Une façon d'assurer l'atomicité d'une diffusion est de permettre une nouvelle diffusion seulement après que la précédente ait terminée, c'est à dire, après que tous les processus du groupe aient reçu le message. Un groupe possède une composition **dynamique** si des membres peuvent être ajoutés/éliminés et **statique** dans le cas où les membres sont permanents. Pour un groupe **dynamique**, le problème est le traitement des messages adressés aux processus qui ont quitté le groupe.

Il est possible de proposer des protocoles plus élaborés que le simple échange de messages entre processus. Les plus connus sont l'**appel de procédure à distance** (RPC) et le modèle "**tableau noir**", popularisé par **Linda**. L'**appel de procédure à distance** (RPC) est un mécanisme qui permet à une procédure d'un processus (le serveur) d'être appelée par un autre processus (le client). Le RPC peut être **synchrone** ou **asynchrone** selon que le client attend ou non la réponse du serveur. Le serveur peut être **séquentiel** si un seul processus traite successivement les requêtes, ou **parallèle** si plusieurs processus sont créés pour traiter les appels en parallèle. Le modèle **Linda** est un modèle "**tableau noir**", c'est à dire, un modèle simplifié de partage de données où un programme est composé de processus et de "tableaux noirs". Les processus communiquent en lisant et écrivant dans un "**tableau noir**". Ce modèle sera détaillé en 2.7.

2.3.3 La synchronisation

Les processus d'une application ont besoin de synchroniser leur actions dans le cas où des propriétés globales de l'état du système doivent être préservées. Par exemple, un contrôle sur l'usage exclusif de ressources communes. Une méthode classique pour préserver un invariant est de garantir l'exclusion mutuelle des accès à des variables d'état et de permettre à un processus de se bloquer en attente de l'occurrence d'une propriété d'état particulière.

Dans les machines UMA/NUMA les mécanismes employées sont, classiquement, les sémaphores et les moniteurs. Les sémaphores sont utilisés pour l'exclusion mutuelle ainsi que pour exprimer la synchronisation. Un sémaphore **s** est une valeur entière qui support deux opérations atomiques : **P(s)** et **V(s)**. L'opération **P(s)** bloque le processus si la valeur du sémaphore **s**, après la décrémentation est inférieure à **0**. L'opération **V(s)** incrémente le sémaphore ; si celui est inférieur ou égal à zéro, il réveille le premier des processus suspendus. La file est généralement gérée dans l'ordre FIFO, préservant ainsi l'ordre des opérations P. Avec les sémaphores, l'utilisateur est le responsable de la synchronisation dans son programme. Les programmes utilisant P et V sont difficiles à lire et prouver.

La notion de moniteur permet de mieux structurer la synchronisation. Un **moniteur** est un module, c'est à dire, une incarnation d'un type abstrait de données encapsulant un ensemble de valeurs qui représentent l'état d'un objet accessible par les seules procédures exportées par le module. Un moniteur est un module particulier dans le sens qu'il garantit l'exclusion mutuelle d'exécution des procédures du moniteur, ainsi que la possibilité, pour un processus, de se suspendre (dans une file d'attente de condition) ou de réveiller un processus suspendu, tout en préservant l'exclusion mutuelle. Les **conditions** permettent d'exprimer des conditions d'attente dépendantes des données du moniteur. Un processus qui fait une opération **wait** sur une condition est bloqué et libère le moniteur. Il sera réveillé par un autre processus par une opération **signal** sur la même condition.

L'implantation efficace d'un sémaphore sur une machine UMA exige un mécanisme élémentaire permettant d'assurer un exclusion mutuelle des accès à la mémoire par les processus. Des instructions effectuent un **test** et **modification** atomique d'une valeur en mémoire commune (ex. TAS sur un verrou). L'existence de niveaux de caches entre processeurs et mémoires exige des algorithmes spécifiques dépendant du protocole de cohérence des caches et dont le but est de limiter l'engorgement du bus. L'exclusion mutuelle dans une architecture parallèle sans cache peut s'exprimer de la façon suivante :

```
/*
```

```
/*
/*    Sans cache
/*
/*
Lock: TAS verrou  | surcharge de la voie
      jnz Lock    | de l'acces a la memoire
    ------------
    - section  -
    - critique -
    ------------
Unlock: verrou = o
```

La suppression de la surcharge du bus peut se faire par une simple temporisation avant les essais. La solution dans le cadre d'une architecture parallèle avec caches et maintien de cohérence forte serait la suivante :

```
/*
/*
/*    Avec cache
/*
/*
Lock:  test verrou | boucle d'attente
       jnz lock    | dans le cache

       TAS verrou  | acces a la memoire (algorithme de maintien de
                   | coherence de cache)
       jnz lock    |
     ------------
     - section  -
     - critique -
     ------------
unlock: verrou = 0
```

La synchronisation dans les machines NORMA ne peut être traité simplement par de tels mécanismes qui reposent sur la mémoire partagée. On ramène les problèmes de synchronisation à des problèmes de communication vers des processus (serveurs) spécialisés dans ce rôle (des arbitres) ou un mécanisme de rendez-vous généralisé (les barrières).

Un programme parallèle est composé par des processus communicants qui s'exécutent sur des processeurs différents. Par exemple, lorsque deux processus P et Q coopèrent dans la résolution d'un calcul de telle façon que le processus Q pour exécuter sa partie a besoin que le processus P lui envoie une valeur, l'opération

receive doit bloquer Q tant que la donnée n'a pas été envoyée par le processus P. Il existe donc une **synchronisation implicite** attachée à la communication. C'est cette synchronisation implicite qui va permettre de bâtir des synchronisations plus complexe. Une manière triviale d'assurer l'exclusion mutuelle d'action est alors de regrouper ces actions au sein d'un processus : le serveur. Tous les autres processus devront demander à ce serveur l'exécution d'une action : ce sont les clients. Le principe du dialogue est que le client émet une demande d'accès et attend une autorisation :

```
send (serveur, action(i), <parametres>) ;
receive (resultat(i)) ;
```

C'est typiquement **appel de procédure à distance** où le serveur joue le rôle d'un moniteur. Le comportement général d'un tel serveur est le suivant :

```
m = receive (any) ;
case (m.action) {
    action1 : ---
              ---
    action2 : ---
              ---
    ...
    actionn : ---
              ---
    }
send (m.demandeur, resultat) ;
```

Une différence est que les moniteurs sont passifs (les procédures sont appelées) et les serveurs sont actifs. Les **serveurs** sont utilisés pour assurer la gestion d'une ressource. Un serveur possède les variables qui définissent l'état d'une ressource et les services qui manipulent cette ressource. Les clients communiquent avec le serveur en demandant un service et reçoivent les résultats. Un serveur sérialise les appels des opérations sur les ressources et assure l'exclusion mutuelle ou la synchronisation entre demandes de services concurrentes. Le serveur peut être réduit au simple rôle de distributeur de droit d'accès. Il est invoqué en tout point d'une action nécessitant une synchronisation avec d'autres actions. C'est un simple arbitre. On écrit trivialement des serveurs sémaphores selon ce modèle. Un problème classique de la programmation distribuée est de passer d'un serveur centralisé (dont l'accès est un goulot d'étranglement) à un service distribué sur plusieurs serveurs coopérants. Ce problème a donné lieu à un grand nombre d'algorithmes de synchronisation distribués.

Les **barrières** représentent un mécanisme de synchronisation utilisé dans le modèle de **processus communicants**. Une barrière est un **rendez vous multiple sans communication**. Les processus qui exécutent l'opération de **barriere(b)** sont bloqués jusqu'à ce que le dernier l'exécute. Les processus continuent alors leur exécution jusqu'à la prochaine barrière. Ce type d'opération est intéressant pour tout calcul parallèle composé de différents phases à enchaîner de façon synchrone.

2.4 Les environnements de programmation

Un **modèle de programmation** doit donc fournir concepts et mécanismes tels que précédement décrits. Ceux-ci se concrétisent généralement dans un langage de programmation et des outils permettant de développer et exécuter ces programmes sur des machines données : un **environnement de programmation parallèle**. Cependant tels environnements ne forment pas un tout monolithique. Nous pouvons distinguer quatre constituants dans un environnement de programmation à destination de machines parallèles : **l'environnement d'évaluation**, **l'environnement de production de programmes**, **l'environnement d'exploitation** et **l'environnement d'exécution**.

Environnement d'évaluation de programmes : il est difficile de prédire les performances d'un algorithme parallèle ainsi que de garantir sa correction. Un ensemble d'outils de preuve, de prédiction de performances et de mise au point constitue **l'environnement d'évaluation**.

Environnement de production de programmes : la production d'un programme parallèle nécessite au moins un langage de programmation et un compilateur. Il est aussi nécessaire de déterminer le **placement** des processus sur les processeurs de la machine parallèle. Ce placement peut nécessiter un regroupement de processus sur de processeurs de façon à obtenir l'accélération maximale.

Dans la pratique, les environnements d'Evaluation et de Production de Programmes sont associés dans un seul système, autour d'un langage de programmation. Par exemple, Start/Pat[App89] est un environnement de développement de

programmes parallèles qui offre des facilités pour la correction et la production de programmes écrits en Fortran. Il est composé d'un **paralléliseur automatique**, un **analyseur statique**, un **correcteur dynamique** et un **analyseur de performances**. Le paralléliseur examine le programme source et peut faire des suggestions pour augmenter le degré de parallélisme ou pour éliminer des constructions qui ne sont pas efficaces. L'analyseur statique simule l'exécution du programme pour détecter des problèmes d'interaction entre les tâches, par exemple l'interblocage. Le correcteur exécute le programme de manière interactive, et l'analyseur de performances collecte les informations données par le système, lors de l'exécution, pour déterminer l'utilisation des processeurs, etc.

Parascope[Cal88] est un projet de développement d'outils d'aide à la conception de programmes parallèles. Cet environnement comprend un **paralléliseur** automatique pour le langage Fortran, un **éditeur** qui permet à l'utilisateur d'exprimer les relations de dépendances de son programme (les systèmes de détection automatique du parallélisme utilisent l'analyse de dépendances pour générer le parallélisme) et un **correcteur de programmes**. L'éditeur Parascope[Ken91] permet au programmeur d'écrire de nouveaux programmes, de convertir de programmes séquentiels en programmes parallèles et d'analyser ces programmes parallèles.

Environnement d'exploitation : la plupart des machines parallèles ne disposent ni de périphériques classique (E/S - disques), ni des services de gestion de programmes et d'usagers permettant une utilisation en exploitation classique. Elles doivent être accédées depuis un système hôte (frontal) pour lequel elles apparaissent comme un périphérique spécialisé, ou depuis un réseau local. Dans ce cas, elles apparaissent comme un serveur spécialisé dans l'exécution de programme parallèle. Un **environnement d'exploitation** a pour rôle la **gestion des accès à la machine parallèle**, l'**allocation des ressources** et le **chargement initial des programmes parallèles**, ainsi que l'**installation des accès du programme parallèle vers l'environnement extérieur**. Le **chargement initial** du programme est la mise en oeuvre d'un **placement**. Certains systèmes utilisent un **fichier de configuration**, défini dans la machine hôte, avec l'association des processus composants le programme aux processeurs de la machine parallèle. Dans ce cas, le **placement** est traité au niveau de l'exploitation. Pour les systèmes sans fichier de configuration, un processus initial

du programme doit assurer le **placement** et le lancement des autres processus.

Environnement d'exécution : c'est la couche logicielle qui supporte le modèle de programme parallèle. Elle est constituée d'un noyau de système sur chaque processeur. Les noyaux coopèrent à la réalisation des services caractéristiques d'un environnement d'exécution pour machines NORMA :

- **placement** et **migration**,
- **communication bi-point** ou **pluraliste**,
- **synchronisation**,
- **gestion de mémoire**,
- **gestion de tâches** et **régulation de charge**.

A ce noyau s'ajoutent les bibliothèques permettant aux programmes parallèles d'accéder aux ressources extérieures (fichiers, entrée/sortie, etc.).

Dans la suite, nous allons présenter quelques environnements de programmation parallèle avant de détailler l'environnement que nous avons défini pour notre machine NORMA : Supernode.

2.5 Le langage Concurrent C

Le langage Concurrent C[Geh86], [Cme89], [Geh92] a été conçu pour la programmation parallèle sur des machines du type UMA puis NORMA. Il utilise un modèle de **processus communicants** et le protocole de communication est de type RPC. Le langage est une **extension du langage C** par des constructions exprimant des exécutions concurrentes ou parallèles : **déclaration** et **création de processus**, ainsi que des opérateurs de **communication**.

2.5.1 Le modèle de programme

Le modèle privilégié de programmation de Concurrent C est le modèle **client/serveur**, basé sur une variante du RPC appelée **transaction**. L'appel d'un serveur

peut être de trois types : **synchrone**, **synchrone avec délai de garde** et **asynchrone**. Dans le RPC **synchrone**, le client qui envoie la requête est bloqué en attente de la réponse. Le RPC **synchrone avec délai de garde** est identique au modèle **synchrone**, sauf que le client continue son exécution lorsque le serveur termine ou lorsque le délai d'attente d'acquisition de sa requête par le serveur est écoulé. S'il reprend son exécution par épuisement du délai, la requête est enlevée de la file du serveur. Dans le RPC **asynchrone**, le client n'attend pas que le serveur exécute la demande, il reprend son exécution après avoir envoyé la requête. Ceci est possible lorsqu'un service n'a pas de résultats. Le RPC **asynchrone** est un simple envoi de message. Ce langage a été créé pour la programmation distribuée, et ne possède pas de mécanismes de programmation concurrente pour le partage de données.

2.5.2 Expression du parallélisme

Le parallélisme en Concurrent C est défini explicitement par l'utilisateur. En Concurrent C, seule une déclaration statique des processus est possible. Un processus ancêtre (main), initialise tous les processus du programme. Ce processus regroupe toutes les déclarations/créations de processus du programme. Ainsi, le nombre de processus du système est défini **statiquement**. Un processus est identifiable par un variable de type **process** qui prend une valeur à la création d'un processus. Un **processus** est créé avec la primitive :

```
process nom ;
nom = create process-model-name (list-parameters)
            [priority (p)] [processor (id)] ;
```

La **list-parameters** spécifie des valeurs initiales pour le processus créé, **priority** la priorité. Le paramètre **processor** définit le processeur où doit être chargé le processus. Si l'utilisateur n'explicite pas ce dernier paramètre, le système démarre le processus sur le processeur le moins chargé. Lorsqu'un processus Concurrent C est créé, il est démarré sur un processeur et reste sur celui-ci jusqu'à la fin : la **migration** n'est pas supportée par le système.

Un processus est défini par une **spécification** et un **corps**. La spécification est la partie publique d'un processus. Cette partie spécification définit la façon dont ont peut communiquer avec le processus. Le corps d'un processus contient son code, les déclarations et les définitions. Le corps n'est pas visible par les autres processus.

2.5.3 La communication et la synchronisation

Le mécanisme de communication entre processus en Concurrent C est l'appel de procédure à distance (RPC) ou **transaction**. Dans une transaction un processus (le client) envoie une demande à un autre processus (le serveur) pour exécuter une opération. Les **transactions** invocables sont définis dans la spécification des processus **serveurs**.

a) **Les clients** : du côté client, l'invocation d'un service (création de transaction) nécessite de pouvoir désigner ce service. Une désignation est une notation qualifiée :

<identification de processus serveur.identification de type de transaction>

Un appel de service ou transaction peut se faire de trois façons :

- **synchrone**
- **synchrone avec un délai de garde**
- **asynchrone**

Dans une **transaction synchrone**, le client attend que le serveur termine l'opération et reçoit toujours une réponse. Si un **temps limite** est spécifié dans un appel à une transaction, le client continue son exécution lorsque l'opération est terminée ou si le **délai d'attente est passé**. Le **délai d'attente** est le temps maximum pendant lequel le client attend que le serveur accepte la requête ; ce n'est pas le délai maximum pour lequel le serveur termine l'opération. Le client ne peut reprendre son exécution à la fin de délai que si le serveur n'a pas encore commencé le traitement de la requête. Une **transaction asynchrone** ne peut ni avoir de **temps limite**, ni retourner des valeurs et le client reprend son exécution aussitôt après avoir envoyé la requête. Un appel **synchrone** se note comme appel de fonction en C :

```
x = fonction-name (arguments) ;
```

où **fonction-name** est une notation qualifiée : **serveur-name.transaction name.** Un appel à une transaction avec un **délai de garde** est dénotée par :

```
within duree ? fonction-name (arguments) : exp
```

- **durée** : temps d'attente en seconds,
- **fonction-name** : identification d'un type de transaction,
- **exp** : si la transaction est acceptée par le serveur dans le délai de temps, la valeur qui revient de la transaction est la valeur de l'expression de l'appel. Sinon **exp** devient la valeur de l'expression d'appel.

Par exemple, soit un serveur **reader** avec un type de **transaction read** sur un périphérique, et un client. Le client envoie un message à l'utilisateur si le périphérique n'est pas accessible avant un certain délai (1 seconde par exemple). L'appel peut être écrit ainsi :

```
r = within 1 ? reader.read () : TIMEOUT ;
if (r == TIMEOUT} printf (`` peripherique n'est pas pret '') ;
```

b) Les serveurs : un **serveur** accepte les **transactions** par la commande **accept**. Elle lui permet de sélectionner parmi les appels d'un même type de transaction ceux à accepter et dans quel ordre. La forme générale de cette commande est :

```
accept transaction-name (parametres) suchthat (tst) by (prty) { action }
```

Les clauses **suchthat** et **by** sont optionnelles et si elles ne sont pas utilisées, l'ordre de réception des **transactions** est FIFO. Si **by** est utilisée, les **transactions** dans la file sont examinées et l'appel où **prty** est la plus basse est exécuté. **prty** est une expression arithmétique qui peut porter sur les variables du serveur ou les arguments de l'appel. Si **suchthat** est spécifiée, seules sont considérées les **transactions** où l'expression conditionnelle **tst** est vraie. S'il en existe plusieurs, elles seront exécutées dans l'ordre spécifié par **by**, sinon dans l'ordre FIFO. L'expression conditionnelle **tst** peut porter sur :

- les variables du serveur,
- les arguments de l'appel.

La commande **select** permet à un serveur de sélectionner une **transaction**. La forme générale est celle d'une commande gardée :

```
select {
     (garde): alternative
     or
     (garde): alternative
     ---
     or
     (garde): alternative
     }
```

Chaque alternative peut avoir une garde qui est une expression logique. qui porte sur les **variables du serveur**. L'ordre d'évaluation est quelconque, et s'il y a plusieurs alternatives exécutables le choix d'une est arbitraire. La commande **select** exécute seulement des alternatives avec la garde vraie ou sans garde. Un processus **serveur** définit les types de **transactions** dans sa partie visible par les autres processus : **la partie spécification**. Un type **transaction** est défini comme une procédure avec le mot clé **trans**. Par exemple, nous présentons ci-dessous la spécification des transactions du processus serveur "**buffer**" (protocole producteur-consommateur).

```
process spec buffer ()
 {
   trans void put (char c) ;
   trans char get () ;
   }
```

Dans cet exemple, le **serveur buffer** déclare deux transactions **synchrones** : **put**, pour mettre un caractère dans un tampon et **get** pour prendre un caractère du tampon. Le mot **void** dans la transaction **put** indique qu'il n'aura pas de paramètre de réponse, et **char** indique le type de réponse de l'opération **get**. La transaction **put** peut être rendue **asynchrone**, si le mot clé **void**

est remplacé par **async** :

```
trans async put (char c).
```

Pour préciser, nous allons présenter un exemple de programme Concurrent C où le serveur **buffer** fait la gestion d'un tampon et deux processus clients, un **producteur** et un **consommateur** l'utilisent. Le serveur déclare (exporte) deux procédures comme visibles par les processus clients, **get** et **put**. Le client **producteur** appelle la procédure **(transaction) put** pour mettre un élément dans le tampon, et le client **consommateur** appelle **get** pour retirer un élément du tampon. Le code d'un client qui produit des éléments dans le tampon est le suivant :

```
process body producteur (process s_buffer)
 {
   int c ;
   for(;;) {
      "obtient la valeur de c"
      s_buffer.put (c) ;
      }
   }
```

Le code d'un client qui consomme des éléments dans le tampon et l'imprime est le suivant :

```
process body consommateur (process s_buffer)
{
 int c ;
 for(;;) {
    c = s_buffer.get () ;
    printf(" Valeur : %d\n", c) ;
    }
 }
```

Le processus **s_buffer** attend toujours une transaction dans la commande **select**. Quand une demande arrive, la garde est évaluée, et si sa valeur est vraie, la transaction est exécutée. Par exemple, si le client **récepteur** exécute la transaction **get** et il n'existe pas d'informations dans le tampon **(n = 0)**, le client reste en attente que la garde devienne vraie (la transaction n'est pas exécutée). Le code du serveur buffer est le suivant :

```
process spec s_buffer () {
    trans int get () ;
```

```
    trans void put (int c) ;
    }

process body s_buffer () {
   int buff_size = 132 ;
   int v ;
   int n = 0 ;
   int in = 0 ;
   int out = 0 ;
   int buff [ buff_size ] ;

   for (;;) {
       select {
            (n < buff_size):
                  accept put (c) {
                       buff [ in ] = c ;
                       in = (in + 1) % buff_size ;
                       n++ ;
                       }
             or
             (n > 0):
                  accept get (c) {
                       v = out ;
                       out = out + 1 % buff_size ;
                       n--;
                       treturn buff [ v ] ;
                       }
             }
       }
   }
```

Les seuls processus composants d'un programme Concurrent C sont créés dans la partie main de ce programme. Dans cet exemple sont créés trois processus, s_buffer, producteur et consommateur.

```
    main()
     {
        process s_buffer s ;
        s = create s_buffer () ;
        create producteur (s) ;
        create recepteur  (s) ;
     }
```

La variable **s** est une variable du type processus, qui contient l'identificateur du processus **s_buffer**. L'opération **create** retourne une valeur qui est l'identification du processus créé. Les processus **producteur** et **consommateur** reçoivent en paramètre l'identificateur du serveur **s_buffer**, ce qui leur permet d'exécuter les opérations déclarées dans la partie **spécification** de ce processus. Les processus sont créés sur le même processeur que "**main**" (l'option processeur de la commande create n'a pas été utilisée). On remarquera que les processus producteur et consommateur sont anonymes du fait qu'ils ne sont serveurs pour personne.

2.5.4 L'environnement de programmation

Il n'y a pas d'outils spéciaux pour la **production de programmes** Concurrent C, ni pour leur **évaluation**. Un programme est créé en utilisant les mécanismes traditionnels offerts par le système Unix. Seul un compilateur Concurrent C permet décrire des programmes parallèles. Cet **environnement de production** est assez fruste, car il est nécessaire de programmer le **placement** et le **lancement** des processus dans le programme Concurrent C lui même.

Dans l'environnement d'exécution pour machine NUMA (section 2.5.5), les caractéristiques de cette machine permettent de vérifier et de modifier les contenus des mémoires. Ceci offre des facilitées de **mise au point** de programmes. Rien de particulier n'est disponible sur la version réseau de Concurrent C. Concurrent C est utilise l'**environnement d'exploitation** fourni par le système de la machine hôte. Ainsi, les fonctions d'entrée/sortie et l'accès aux fichiers sont celles du système Unix.

2.5.5 L'environnement d'exécution

Le compilateur Concurrent C produit du C. A tout corps de processus correspond une procédure C et à tout processus correspondra une pile propre pour ses variables locales et appels de procédures. Pour chaque **transaction**, le compilateur génère un appel à une fonction (stub) qui fabrique un message dont les composants sont les arguments de la transaction (emballage). L'envoi et la réception de messages sont des appels au noyau parallèle. Pour chaque accept/select, le

compilateur génère l'enchaînement de consultations des files de message (appel au noyau). Le rôle du noyau exécutif est d'assurer la **création** et le **placement** des processus ainsi que la **communication** et la **synchronisation.**

Création et placement : un programme Concurrent C a un nombre statique connu a priori de processus dont le lieu de placement est fixé au départ.

Communication et synchronisation : Les étapes d'exécution d'une **transaction synchrone** sont les suivantes :

1. Si le serveur est local, mettre le message dans la file du processus serveur, réveiller le serveur (si nécessaire) et se bloquer en attendant la fin de la transaction.
2. Si le serveur est distant, le message est envoyé au processeur concerné où il doit exister un processus qui traite les messages distants. Ce processus reçoit le message, le met dans la file du serveur et réveille celui-ci, si nécessaire.
3. Le serveur enlève la description de la transaction de la file et l'exécute.
4. Quand le serveur termine il envoie les résultats au processus (local ou distant) qui a demandé la transaction, et le client est réveillé.

Une **transaction asynchrone** est réalisée de la même façon, sauf que le client ne reste pas bloqué en attendant la réponse. Le principe d'exécution d'une **transaction avec délai de garde** est aussi le même, mais le client peut reprendre son exécution à la fin du délai d'attente, après avoir demandé le retrait de sa requête au serveur.

Concurrent C a été implanté sur une machine NUMA et une machine NORMA[Geh92].

NUMA : cette machine est composée par un ensemble de processeurs et mémoires connectés sur un bus commun avec une machine hôte Unix. Chaque carte possède un processeur et de la mémoire locale. Cependant un processeur voit une mémoire globale unique : il peut référencer la mémoire globale et les mémoires locales des autres processeurs. Sur chaque processeur est chargé un noyau qui est capable de créer, exécuter et terminer des processus. C'est un noyau qui supporte

la multiprogrammation et tous les processus dans ce noyau partagent le même espace d'adressage. Les processus communiquent en utilisant des structures de données placées dans la mémoire partagée. Il existe un mécanisme d'exclusion mutuelle pour synchroniser les accès aux données partagées.

Concurrent C est implanté au dessus de ce noyau et chaque processus Concurrent C est un processus géré par ce noyau. La mémoire partagée est utilisée pour passer les arguments d'un appel de procédure à distance et pour retourner les réponses. Le tableau de descripteur des processus est aussi dans la mémoire globale.

La réalisation d'un appel RPC est fait de la façon suivante : le client obtient l'accès au tableau des descripteurs (en exclusion mutuelle), met le message dans la file du serveur, réveille le serveur, libère le tableau des descripteurs et se bloque. Le serveur enlève la demande de la file, exécute l'opération et réveille le client. Le client passe les adresses des arguments et les adresses où le serveur doit mettre les valeurs de réponse dans son descripteur.

NORMA : la version NORMA est réalisée sur des ordinateurs VAX (sous système Unix) connectés sur un réseau Ethernet. Chaque programme Concurrent C est composé d'un ou plusieurs processus Concurrent C. Un programme Concurrent C est implanté par un ou plusieurs processus Unix qui nous appelerons processeurs virtuels pour limiter l'ambiguité due à l'usage du terme processus dans deux contextes différents. Ces processeurs virtuels sont connectés par un réseau. Les processeurs virtuels exécutent tous le même code ; le programme Concurrent C est **répliqué** sur les processeurs virtuels. Un noyau de processus léger (thread) assure l'exécution des processus Concurrent C. Le programme Concurrent C constitue un processus **lourd** (tâche à la Mach[Acc86]) et chaque processus Concurrent C est implanté par un processus léger (thread à la Mach). Un serveur de communication réseau permet à un processus Concurrent C d'appeler un serveur situé sur un autre "processeur virtuel". Ce serveur transmet la requête via le réseau (protocole TCP/IP[Ste90], [Tan90]) à un serveur identique sur le site distant. Celui appelle le serveur invoqué et transmet la réponse au serveur initial, qui à son tour répond au client. Ainsi, quand un processus Concurrent C, **p** appelle un autre processus **q**, si **p** et **q** sont sur le même processeur virtuel, le noyau exécutif

local met la requête dans la file de messages de **q** et le processus **p** est bloqué. Lorsque **q** a traité la demande, il réveille **p**. Si **p** et **q** sont sur des processeurs virtuels différents, le message de **p** est envoyé par le serveur de communication local au serveur de communication du processeur contenant **q**. La réponse de **q** reçoit un traitement identique et est transmise à **p**. Ainsi, l'échange de messages entre les processus distants se passe de manière transparente à l'utilisateur.

Cependant, il est à la charge du programmeur d'écrire le programme de lancement, c'est à dire la création des processeurs virtuels et des processus tournant sur ces processeurs virtuels. Les processus Concurrent C sont créés par l'utilisateur sur un processeur virtuel. Le lancement des processus Concurrent C est fait par le processus **main**, qui s'exécute seulement sur le processeur virtuel dans lequel l'exécution du programme Concurrent C a commencé. Les processus virtuels doivent être créés préalablement au lancement des processus par une commande :

c_processor (machine, program)

2.6 Le langage Orca

Le langage Orca[Bal90], [Bal92], [Tan92], est un langage de programmation parallèle pour machines NORMA. C'est un **nouveau langage** qui a été conçu pour être simple à utiliser et pour qu'il soit possible, lors de la compilation, de détecter des erreurs de type dans la communication. Les processus coopèrent en utilisant des données partagées, même s'ils sont sur des processeurs différents. Les données partagées sont manipulées par des opérations définies par l'utilisateur.

Une manière de programmer en Orca est de répliquer un même programme sur un ensemble de processeurs. Chaque processeur exécute un certain nombre de **processus**. Un processus "main" commence sur un processeur et démarre à son tour d'autres **processus** localement ou à distance.

2.6.1 Le modèle de programme

Le modèle Orca est un modèle où les processus opèrent sur des **objets partagés**. Ceux ci sont accédés par l'invocation d'opérateurs caractérisant le type de l'objet. On peut considérer que l'on a une forme de **modèle parallèle à partage de données**. Dans ce modèle, le programmeur peut utiliser des données partagées même si la machine ne possède pas de mémoire physique partagée. Les données partagées sont manipulées par des opérations de haut niveau, définies par l'utilisateur. Le modèle Orca est identique au modèle Concurrent C, mais il n'y a plus un serveur centralisé et unique par donnée partagée ; il est **dupliqué** de façon non limitative sur les processeurs. En conséquence :

a) les appels RPC deviennent un appel à un serveur local,

b) le serveur local invoqué effectue l'action sur la copie locale de l'objet puis diffuse ces modifications aux autres serveurs avant de valider la modification locale. La **cohérence des réplications** est garantie par l'atomicité de la **diffusion** de mise à jour en fin d'opération, ce qui assure une cohérence séquentielle (non forte) des modifications. Les opérations parallélisées entre réplicas donnent un gain d'efficacité, dès que le parallélisme est possible par partition de données en objets indépendants ou par parallélisation des accès en lecture à un objet.

2.6.2 Expression du parallélisme

Le parallélisme dans Orca est défini explicitement par la création d'un processus. Un processus en Orca est défini comme suit :

```
process name (parametres)
   declarations des variables locales
begin
   commandes
end;
```

Au début, un seul processus du programme est actif. De nouveaux processus peuvent être créés par la commande :

```
fork name (parameters) [ ON (cpu number)]
```

Cette commande crée **dynamiquement** un nouveau processus local ou distant, qui à la différence de Concurrent C est toujours anonyme. La partie **ON** de la commande définit le processeur. Si elle n'est pas utilisée, le processus sera créé sur le processeur du père. Le système ne fait pas de **régulation de charge**. Les processus qui sont **démarrés** sur un processeur restent sur ce processeur jusqu' à la fin (**pas de migration**). Les paramètres peuvent être de deux types : **input** (par valeur) et **data-objects** (par référence). Dans le cas des paramètres input, le processus reçoit une copie d'un objet. La valeur ceux-ci peut être une structure de données quelconque.

2.6.3 La communication et la synchronisation

Le seul moyen de communication entre processus provient du partage des objets.

a) **Objet partagé** : un **data-object** est une incarnation d'un type abstrait de données décrit par une **spécification** et une **implantation**. La partie **spécification** définit les opérations qui seront exécutées sur l'objet. La partie **implantation** est composée des données qui représentent l'objet, par le code d'initialisation des données et du code des opérations sur les données. Un objet est créé de manière **explicite** et **dynamique** par la déclaration d'une variable de type d'objet. Une zone de mémoire est allouée pour les données de l'objet puis le code d'initialisation est exécuté. Un exemple de programme est le suivant :

```
object specification compteur ;
       operation incrementer (a : integer) ;
end;

object implementation compteur ;
    a : integer ;
    operation incrementer (v : integer) ;
    begin
         a = a + v ;
         return a ;
    end ;

 begin
      a = 0 ;
 end ;
```

Dans cet exemple, un type objet, **compteur**, a été déclaré. Il comporte la définition de l'opération **incrementer** qui peut être effectuée sur l'objet. La partie implan-

tation contient le code de l'opération, la déclaration de la variable **a** et le code d'initialisation de la variable **a** .

b) **Coopération** : la communication provient du partage d'objets. Un processus invoque un opérateur d'un objet. C'est un appel assimilable à un RPC **synchrone**, c'est à dire que le processus invocateur ne reprend en séquence que après l'exécution complète de l'opération. Ainsi, dans l'exemple suivant :

```
value : integer ;
s : compteur ;

value = s$incrementer (5) ;
```

le processus exécutant ce code applique l'opération **incrementer (5)** sur l'occurrence **s** d'un **compteur**. La seule possibilité de **coopération** entre processus provient donc du partage des données par le biais des objets partagés.

c) **La synchronisation** : Orca offre deux niveau de synchronisation : le premier niveau s'intéresse à mantenir cohérent la valeur d'un objet lors d'accès concurrents. L'exclusion mutuelle a l'avantage de la simplicité mais restreint trop le parallélisme d'accès aux objets. Un protocole lecteur/rédacteur autorisant les lectures en parallèle est plus intéressant, tout en garantissant l'existence d'une seule valeur "visible" de l'objet. Orca a preféré accroître encore les possibilités de parallélisme en proposant un protocole autorisant en parallèle des lectures (de la valeur valide) et une écriture (de la nouvelle valeur). Ce protocole "multiple lecteur et un rédacteur" est dit à cohérence faible. Pour implanter ce protocole Orca :

- classe les opérations selon qu'elles modifient ou non la valeur de l'objet
- autorise une écriture en parallèle de plusieurs lectures, c'est à dire, la construction d'une nouvelle valeur avec la consultation de l'ancienne (cohérence faible).

La seconde forme de synchronisation permet de bloquer l'exécution d'une opération tant qu'une condition n'est pas satisfaite. Les processus syncronisent donc de manière implicite par l'invocation d'opérations sur des objets partagés. Une opération bloquante est composée par un ou plusieurs commandes gardées.

```
operation op (parametres) : result ;
```

```
begin
     guard condition1 do
          commandes ;
     od
     ---
     guard conditionn do
          commandes ;
     od
end ;
```

Les conditions sont des expressions booléennes, qui peuvent dépendre des paramètres d'appel, des données de l'opération et des données de l'objet.

Orca et Concurrent C sont très voisins. A titre d'exemple, il est possible d'écrire en Orca l'exemple Concurrent C présenté en 2.5.3.

```
object specification s_buffer ()
  operation int get () ;
  operation void put (c : integer) ;
end ;
object implementation s_buffer ()
   int buff_size = 132 ;
   int v = 0 ;
   int n = 0 ;
   int in = 0 ;
   int out = 0 ;
   int buff [buf_size] ;

      operation  put (c : integer)
         begin
             guard (n < buf_size) do
                buff [ in ] = c ;
                in = (in + 1) % buff_size ;
                n++ ;
             od
         end

      operation get ()
         begin
             guard (n > 0) do
                v = out ;
                out = out + 1 % buff_size ;
                n--;
```

```
            return buff [ v ] ;
        od
    end

  begin
  end;
```

Le code d'un client qui produit des éléments dans le tampon est le suivant :

```
process producteur (buffer: shared s_buffer)
  {
   int c ;
   for(;;) {
       "obtient la valeur de c"
       buffer$put (c) ;
       }
   }
```

Le code d'un client qui consomme des éléments dans le tampon et l'imprime est le suivant :

```
process consommateur (buffer: shared s_buffer)
  {
   int c ;
   for(;;) {
       c = buffer$get () ;
       printf(" Valeur : %d\n", c) ;
       }
  }
```

Le code du processus "main" est le suivant :

```
process Orcamain()
  buf: s_buffer ;
  begin
      fork producteur (buf ON 1) ;
      fork recepteur  (buf ON 2) ;
  end;
```

L'exécution du programme montré ci-dessus débute par le processus **Orcamain**, qui déclare une variable **buf** du type **s_buffer** (objet partagé). Il crée ensuite deux processus **producteur** et **récepteur** sur les processeurs 1 et 2 et passe comme paramètre l'objet partagé (la variable buf).

Dans le cas du programme Orca, deux copies de l'objet existent et sont alternativement accédées par les processeurs 1 et 2. Dans le cas du programme équivalent Concurrent C, les deux processus auraient accéder le serveur représentant le seul objet **buffer**. Réciproquement, il est possible d'exprimer un objet Orca comme un serveur Concurrent C. Les tableaux 2.1 et 2.2 résument les differences entre Concurrent C et Orca. Cependant, la plus grande difference provient de la pos-

Aspect	Concurrent C	Orca
Variables d'état	oui	oui
Paramètres	oui	oui
Priorité	oui	non

Table 2.1: Puissance relative des gardes de Concurrent C et Orca.

Invocation	Concurrent C	Orca
Synchrone	oui	oui
Timeout	oui	non
Asynchrone	oui	non

Table 2.2: Types d'appels des services.

sibilité en Orca de **créer dynamiquement des objets** et des **processus** (tableau 2.3).

Dynamicitée	Concurrent C	Orca
Processus anonyme	non	oui
Objets/serveur	non	oui

Table 2.3: Création dynamique des entités.

2.6.4 L'environnement de programmation

L'environnement de production de programmes comprend le compilateur Orca. Il n'existe apparemment pas d'outils de prédiction de performances ou de mise au point. Aucun outil de placement n'est disponible et le programmeur doit exprimer lui même le placement des processus ; la migration n'est pas possible. Orca utilise l'**environnement d'exploitation** fourni par le système de la machine hôte.

2.6.5 L'environnement d'exécution

L'implantation s'est intéressée à l'exploitation maximum des possibilités de parallélisation liés :

- à la création dynamique de processus et objet,
- à la définition d'une cohérence faible des objets permettant des accès, parallèles en écriture et modification.

Le principe d'implantation est la **réplication totale** des code et données (objets) sur les processeurs, impliquant que :

- un processus peut être créé n'importe où,
- un objet est répliqué sur tous les processeurs, qui peuvent l'accèder directement.

Le problème principal de l'implantation est celui du maintien de la cohérence des copies multiples des objets. Le principe est le suivant : toute modification est faite à partir de la copie locale au processeur et produit une nouvelle valeur qui est diffusée à tous les processeurs (y compris le processeur effectuant la modification). La valeur diffusée devient la nouvelle valeur des copies de l'objet. La cohérence des copies est assurée par l'utilisation d'une **diffusion atomique** qui garantit que tous les processeurs "voient" la même suite de valeurs (cohérence séquentielle).

La duplication est utilisée pour diminuer le temps d'accès aux données partagées. Chaque processeur préserve une copie des données partagées qui sont accessibles par tous les processus d'un processeur. Les opérations qui ne modifient pas les données sont exécutées en parallèle et utilisent la copie directement. Les opérations modifiant la valeur diffusent la nouvelle valeur sur tous les processeurs. L'implantation est divisée en trois niveaux :

1. **La compilation.**
2. **Le noyau d'exécution.**
3. **La diffusion fiable.**

- **La compilation** : les programmes Orca sont traduits en un code contenant les appels au noyau d'exécution parallèle Orca. Une phase terminale du compilateur analyse le code engendré pour chaque opération afin de déterminer celles qui modifient les données partagées. Ceci permet de savoir si une diffusion des modifications sera nécessaire. Cette information est mise dans un **descripteur d'opération** accompagné d'informations sur la taille et le type (entrée/sortie) des paramètres. Pour chaque invocation d'une opération sur un objet, le compilateur génère un appel au noyau exécutif Orca par la la primitive : **invoke (objet, descripteur d'opération, paramètres)**.

- **Noyau exécutif** : ce niveau est responsable de la gestion des processus et objets dupliqués. Il implante la primitive **invoke**. Un programme Orca est repliqué sur tous les processeurs comme un processus lourd (tâche). Tout processus Orca est implanté comme un processus léger. Un processus léger particulier, le **gestionnaire d'objets** est responsable sur chaque processeur de l'actualisation de toutes les copies des objets partagés sur le processeur. Les objets et leurs copies sont placés dans une partie de la mémoire accessible par le **gestionnaire d'objets**. Les processus peuvent lire les objets sans l'intervenction du **gestionnaire d'objets**. Par contre, si un processus veut effectuer une opération qui modifie l'objet il transmet cette opération au **gestionnaire d'objets** et se bloque. Toute nouvelle valeur d'objet doit être actualisée sur tous les processeurs. Deux solutions sont possibles : soit diffuser l'opération à tous les sites, soit calculer la modification (simuler l'opération) et diffuser cette modification à tous les sites. L'atomicité de la diffusion doit garantir que tous les sites y compris les initiateurs voient les invocations ou modifications dans la même ordre. Chaque **gestionnaire d'objets** enregistre ces diffusions dans une file FIFO et les traitent de la façon suivante :

  - la modification est enlevée de la file.
  - la copie de l'objet est bloquée.
  - la mise à jour est effectuée.
  - la copie est libérée.

- **La diffusion atomique fiable** : la diffusion atomique est faite de façon indirecte. Le gestionnaire d'objet transmet à un processus particulier d'un site donné

le message à diffuser. Ce processus, dit **séquenceur** reçoit les demandes de diffusion dans un ordre donné. Il affecte un numéro à chacune d'elles et les réalise, l'une après l'autre. La fonction de **séquenceur** est attribuée à un noyau exécutif Orca arbitraire au démarrage du programme.

2.7 Le langage Linda

Linda est constitué d'un ensemble de primitives qui permet d'étendre un langage séquentiel en un langage de programmation parallèle. Dans le modèle Linda, les processus communiquent **indirectemant** par l'inscription ou la consultation de données ou tuples sur un "tableau noir" (un espace de tuples). Un langage Linda est simplement l'extension d'un langage quelconque par des primitives permettant de **placer**, **d'enlever** et **lire** un tuple dans l'espace de tuples ainsi que de **créer un processus**.

2.7.1 Le modèle de programme

Linda offre un modèle de programmation parallèle à partage de mémoire mais où la mémoire se réduit à une forme particulière et restreinte de **mémoire associative partagée**. Un processus produit des données et les place dans cet espace par des opérations **out**. Un processus qui a besoin d'une de ces données peut la consulter par une opération **in** ou **read**. Les données sont des paires **(étiquette, valeur)** inscrites sans ordre particulier dans l'espace des tuples. Cet espace est une **mémoire associative** dans la mesure où l'on retrouve un tuple particulier en utilisant son champ étiquette comme clé d'accès. Cette recherche effectue un filtrage de l'espace de tuples par l'étiquette.

2.7.2 Expression du parallélisme

La seule primitive de création d'un processus est l'opération **eval**. Le paramètre du **eval** a un format de tuple. La partie **valeur** est la description du processus à lancer **dynamiquement** et son interpretation est laisée au système hôte sous-jacent. La clé identifie le processus et la valeur restituée par celui-ci à la fin de son exécution. Par exemple, dans la commande :

```
eval("t", compute (a,b)) ;
```

compute (a , b) est l'invocation d'une fonction. Un processus est créé pour exécuter "**compute (a,b)**". Quand ce processus termine, il dépose le résultat de la fonction comme un tuple d'étiquette "**t**" :

```
("t", resultat) ;
```

Linda ne posséde aucun mécanisme qui permet de spécifier le **placement**. Si celui c'est possible, c'est parce que le système hôte le permet.

2.7.3 La communication et la synchronisation

Les processus communiquent par l'intermédiaire de l'espace de tuples. Les tuples sont des valeurs identifiées par une clé et sont manipulés par les opérations suivantes :

```
out (cle, valeur)
in (cle, variable)
read (cle, variable)
```

Les opérations d'accès **out**, **in** et **read** sont réalisées de façon non controlée. Une opération **out** ajoute un tuple dans l'espace de tuples. L'opération **out** est non bloquante et si des **outs** successifs sont effectués avec la même clé, ils seront enregistrés dans l'espace de tuples dans un ordre quelconque. L'opération **in** recupère un tuple de clé donnée, affecte la valeur du tuple à la variable argument et l'enlève de l'espace de tuples. S'il existe plusieurs tuples de même clé le choix est arbitraire. L'opération **read** est similaire à l'opération **in** mais le tuple reste dans l'espace de tuples. Les opérations **in** et **read** sont bloquantes s'il n'y a pas de tuple de clé cherchée. Si deux processus exécutent la commande **in** sur un tuple unique, un seul possèdera le tuple (atomicité). L'autre sera suspendu, mais on ne peut pas savoir a priori qui obtient le tuple. Dans le cas où un processus effectue un **read** et un autre **in**, si le **read** est exécuté avant, le **in** est aussi exécuté. Dans le cas contraire, si le **in** est exécuté avant, le processus qui fait le **read** est suspendu jusqu'à ce qu'un nouveau tuple, de même clé soit disponible dans l'espace de tuples. La communication est donc **indirecte**. Les processus communiquent de façon anonyme, c'est à dire, sans se connaître. Seule la connaissance des clés est nécessaire. S'il y a plusieurs tuples de même clé (au sens adresse de mémoire), un quelconque parmi eux est pris. Le problème qui se pose est alors de savoir comment un dialogue particulier peut s'établir entre processus. L'échange de messages entre **deux processus** est realisé par les opérations **in** et **out** via une clé convenue. Par exemple, l'opération

```
out ("a", 57) ;
```

place la tuple de clé "a", avec la valeur 57 dans l'espace de tuples. L'opération

```
int i ;
in ("a", i) ;
```

récupere un tuple de clé "a" et dont la valeur est un entier. L'opération

```
int i, j ;
in ("a", i, j) ;
```

démande un tuple de clé "a" avec deux champs entiers. Nous pouvons aussi filtrer la valeur du tuple par des champs constants.

```
int j ;
in ("a", 5, j) ;
```

Dans ce cas, le numéro 5 fonctionne comme une clé secondaire. Cette opération récupere un tuple qui posséde la clé "a" et la valeur 5 dans le premier champ.

La **diffusion** peut être vue comme une combinaison d'un **out** et de **read** des processus destinataires. Par exemple, si on considére un processus qui produit une nouvelle valeur pour un tuple **v** et qui doit l'envoyer à un ensemble de processus, il exécute l'opération **out ("v", valeur)**. Les processus concernés peuvent le recevoir avec **read ("v", variable)**. Comme le tuple n'est pas enlevée par un **read**, tous les processus qui participent à la **diffusion** accèdent à la nouvelle valeur. L' ensemble des processus qui participent à la **diffusion** peut être un sous ensemble des processus existants (**diffusion partielle**) ou tous les processus (**diffusion totale**). Le problème est alors de savoir quand retirer la tuple diffusé. Il est nécessaire pour cela que les récepteurs acquittent leur réception permettant ainsi à l'initiateur de retirer le tuple diffusé de l'espace de tuples.

On peut simuler un RPC en utilisant une combinaison des opérations **in** et **out**. Le processus client peut faire un appel par :

```
out ("S", "client", parametres) ;
in  ("client", reponse) ;
```

Le paramètre d'appel **client** sera utilisé comme clé du tuple qui contient les résultats de l'opération et **paramètres** contient les informations nécessaires pour l'exécution de la procédure. Après avoir envoyé la demande, le processus est suspendu (par l'opération in) en attendant la réponse. Un serveur est représenté par une boucle du type :

```
serveur ()
 {
  for(;;) {
```

```
    in  ("S", "client", variables) ;
     -
     -
    out ("client", valeurs) ;
    }
  }
```

Le serveur utilise l'opération in pour recevoir les appels, et après avoir exécuté l'action indiquée par les paramètres, envoie la réponse par une opération **out**. Le serveur peut entrelacer plusieurs requêtes à son gré ou plutot à celui du programmeur. Cependant, la sémantique du in interdit un service FIFO équitable. Si la procédure appelée est une fonction pure, on peut faire un **eval** suivi d'un in.

```
eval ("nom", f (x, y) ;
   ---
   ---
in  ("nom", valeur) ;
```

Selon que le client attend le résultat immédiatement après ou non, on a un RPC **synchrone** ou **asynchrone**. L'exclusion mutuelle d'accès à une donnée est trivialement réalisable. Une valeur partagée par plusieurs processus est mise dans un tuple et si des processus exécutent l'opération in ("**mutex**", **variable**) en même temps, un seul va réussir à lire le tuple. Le processus qui a obtenu le tuple le remet dans l'espace de tuples avec l'opération **out**("**mutex**", **v**) et débloquera ainsi un autre demandeur. Le problème de cette solution est la famine. Comme les opérations in sont réalisées de manière arbitraire, est possible qu'un processus ne récupère jamais les données.
Une **exclusion mutuelle** booléenne se réduit à :

```
  in ("mutex") ;

     "section critique"

  out ("mutex") ;
```

Comparer Linda à Orca ou Concurrent C est difficile du fait de la "faible sémantique" de Linda. La propriété de choix non deterministe de tuples de même clé pose des problèmes pour écrire un serveur n'introduisant pas de famine ou une diffusion fiable. Par ailleurs, le choix de clés convenues entre processus joue un rôle important et rien ne vient aider à leur conception. Une autre façon de faire est de proposer une implantation de Linda en Concurrent C, sous forme de serveur centralisé du bien en Orca sous forme "tableau noir". On peut programmer un espace de tuples Linda avec un serveur centralisé, en Concurrent C de la façon suivante :

```
process spec s_tuples ()
{
  trans val-tuple  in (type-cle k) ;
  trans void out (type-cle k, val-tuple v) ;
  trans val-tuple read (type_cle k) ;
}

process body s_tuples
 for(;;) {
     select{
          accept in (k) suchthat(existe-tuple(k))
                "chercher un tuple <k, v> de cle k
                et l'enlever de l'espace de tuples"
                t-return (v) ;
          or
          accept read (k) suchthat(existe-tuple(k))
                "le tuple n'est pas enleve de l'espace de tuples"
                t-return (v) ;

         or
         accept out (k, v)
                "mettre <k, v> dans l'espace de tuples"
         }
     }
```

Les opérations **int** et **out** utilisent une garde. Elles seront exécutées seulement s'il existe un tuple avec la clé spécifiée. Le tuple retourné dans l'opération **in** est enlevé de l'espace de tuples. Dans le cas de l'opération **read**, il continue a exister dans l'espace de tuples.

Dans le langage Orca, nous pouvons programmer un espace de tuples Linda d'une façon voisine :

```
object specification s_tuples ()
  {
  operation void in (type_cle k, type_val v) ;
  operation void out (type_cle k, type_val v) ;
  operation void read (type_cle k, type_val v) ;
  }

object implementation s_tuples ()
      operation  in (k, v)
         begin
```

```
        guard(existe-tuple(k))
          do
            "copier la valeur du tuple trouve dans v
            enlever le tuple de l'espace de tuples"
          od
        end

    operation read (k, v)
      begin
        guard(existe-tuple(k))
          do
            "copier la valeur du tuple trouve dans v"
          od
        end

    operation out (k, v)
      begin
         "mettre <k, v> dans l'espace de tuples"
      end

    begin

    end;
```

L'implantation Concurrent C est un implantation centralisée où tous les accès sont distants et sérialisés. L'espace de tuples est un goulot d'étranglement du système, le taux d'accès est élevé. Orca donne une implantation privilégiant la parallélisation des accès au prix d'une réplication complète de l'espaca de tuples et d'une diffusion atomique comme opération de communication de base.

2.7.4 L'environnement de programmation

Pour la **programmation parallèle**, Linda offre les seuls mécanismes de communication et de synchronisation présentés précedemment. Linda se réduit à une bibliothèque parallèle appelable d'un langage quelconque et utilisant les fonctions du système hôte. Aucun compilateur particulier n'est nécessaire. L'environnement Linda ne propose aucun outil pour **l'évaluation de programmes** et pour **l'exploitation de la machine parallèle**. Il repose integralement sur les possibilitées du système

hôte.

2.7.5 L'environnement d'exécution

L'implantation de Linda a été étudiée dans le cadre d'environnements à mémoire partagée (machines UMA/NUMA) ou à mémoire distribuée (machines NORMA). Dans le modèle à mémoire partagée, les opérations **in**, **out** et **read** travaillent sur un espace de tuples qui est placé dans une zone de mémoire commune à tous les processus.

Dans les systèmes à mémoire distribuée (NORMA), le problème principal est la distribution de l'espace de tuples sur les différentes mémoires. Ce problème se divise en deux parties : comment stocker les tuples et comment les localiser. Plusieurs solutions sont possibles selon le degré de distribution choisi :

1. Un **serveur centralisé** gére l'espace de tuples et un seul noeud sert au stockage de ces tuples (**espace de tuples centralisé**). Les opérations **in**, **out**, **read** génèrent des requêtes à ce serveur. Cette soluction est simple, mais le serveur constitue un point critique du système. Il sera peut être surchargé par la quantité de messages à traiter, sa mémoire peut ne pas pouvoir stocker tous les tuples du système (espace theoriquement limité). Il est, en effet, à la charge du programmeur de retirer les tuples devenus inutiles.

2. Les tuples sont stockés sur les noeuds où ils sont engendrés par une opération **out** et un serveur centralisé maintient un **tableau de clés de tuples**, associant à une clé les noeuds où une valeur de même clé est stockée. Une opération **in** ou **read** se traduit en une requête d'interrogation de ce serveur pour connaître la **localisation de la valeur** suivi d'une requête de consultation ou de retrait pour le noeud qui contient le tuple. Une opération **out** engendre une requête d'enregistrement de la clé et de la valeur. Cette solution élimine le problème de stockage centralisé des valeurs, mais ne réduit pas le flot de messages vers le serveur de localisation, qui constitue toujours un goulot pour le système en raison du nombre de communications qu'il doit traiter.

3. Les tuples sont stockés sur les noeuds où ils sont générés par l'opération out. Chaque fois qu'un nouveau tuple est généré sur un noeud, sa clé et sa

localisation sont diffusées à tous les noeuds existants (**diffusion globale**). Les opérations **in** et **read** consultent localement et une seule requête est envoyée pour obtenir la valeur. L'inconvénient de cette solution est la duplication du tableau des couples **<clé, localisation>** sur chaque noeud du système.

4. Les tuples sont gérés localement sans serveur de localisation local ou global. Les opérations **in** et **read** doivent consulter tous les sites pour trouver un tuple (**diffusion**). L'espace de tuples est partitionné sans rédondance.

Une implantation de la machine virtuelle Linda pour une machine NORMA est proposée dans [Gel85]. Dans cette machine virtuelle, le **tableau de clés** est **distribué** sur les processeurs et les valeurs de tuples sont stockées sur les processeurs où ils sont générés (**espace de tuples distribué**). La distribution de l'espace de tuples est faite de façon à n'impliquer qu'une diffusion partielle des requêtes de modifications (**out**) ou de consultation (**in**, **read**). Pour ce faire, on plaque une géométrie de grille rebouclée (2D) sur les processeurs. Tout noeud appartient donc à une ligne et une colonne de la grille. Chaque colonne maintien une réplication de la table des clés engendrée par un noeud de la colonne. Ainsi, tout **out("k", valeur)** fait par un processeur **<l, c>** provoquera une diffusion sur la colonne **c** du message :

"le processeur <l, c> a une valeur de la clé k".

Toute l'opération **in("k", ..)** ou **read("k", ..)** fait par un processeur **<l, c>** fera une diffusion sur la ligne "l" de la requête

"qui posséde une valeur avec la clé k".

Un processeur ou plus de la ligne "l" répondra **<c, l> a un tuple de clé "k"**.
Une diffusion partielle n'implique que $N^{\frac{1}{2}}$processeurs (N est le nombre total de processeurs). Outre cet intérêt immédiat, la géométrie de distribution choisie s'applique idéalement à une architecture maillée de type grille 2D, comme un réseau de Transputers. La diffusion sur une ligne ou colonne est trivialement assurée par un jeton circulant sur la ligne ou la colonne.

2.8 p4 Programming System

Comme Linda, le **p4 Programming system** se présente comme un ensemble de procédures de bibliothèque et de macros pour la programmation parallèle et distribuée, utilisable depuis n'importe quel langage (ex. C, Fortran). Il a été développé à l'Argonne National Laboratory[But92]. Ce système a été conçu pour être portable sur plusieurs types de machines parallèles. Il a été adapté à des réseaux d'ordinateurs homogènes ou non (NORMA), comme à des machines multiprocesseurs avec mémoire commune (UMA/NUMA).

2.8.1 Le modèle de programme

Le système supporte deux modèles de programmes : **mémoire partagée** et **échange de messages**. Le modèle à **mémoire partagée** est basé sur des **moniteurs**. Le modèle à **échange de messages** intègre des primitives de communication du type **send**, **receive** et de **diffusion** ainsi qu'un mécanisme de **barrière** pour la synchronisation.

2.8.2 Expression du parallélisme

Dans le modèle à **échange de messages** (machine virtuelle NORMA), le programmeur **place statiquement** un ensemble de processus lourds soit sur une machine à **mémoire partagée (UMA)** soit sur une machine à **mémoire distribuée (NORMA)**. Un fichier, appelé **procgroup**, décrit le placement de ces processus lourds. Chaque entrée dans ce fichier contient :

- le nom de la machine où le **processus lourd** doit être exécuté,
- le nombre de **processus lourds** (occurrences du même processus lourd) qui doivent être créés,
- le nom du fichier qui contient le code.

La procédure **p4_create_procgroup()** permet démarrer le programme parallèle. Cette primitive accède au fichier qui décrit l'ensemble des **processus lourds** à créer et installe un réseau de processus lourds ayant chacun une identification unique gérée par le système. Ils communiquent par **échange de messages.**

Des processus légers sont créés de façon **dynamique** et **explicite** par au sein d'un processus lourd par :

```
p4_create (addr)
```

L'argument **addr** est une fonction. Tous les processus légers d'un processus lourd partagent la mémoire de ce processus lourd. Les problèmes de communication et synchronisation sont assurées par des **moniteurs**. Par contre, la communication entre processus lourd n'est possible que par l'intermédiaire de messages.

p4 combine donc les modèles **à mémoire partagée** et à **processus communicants**. Par exemple, on peut utiliser **p4_create_procgroup()** pour créer un réseau de processus qui communiquent par **échange de messages** (machine virtuelle NORMA), et chaque processus peut créer des processus légers, avec la primitive **p4_create (addr)**, qui communiquent par la **mémoire partagée** (machine NORMA).

Cependant le système p4 ne supporte ni **création dynamique** de processus lourds, ni **création à distance** de processus légers, ni la **migration** de processus lourds/légers. Il ne fait pas de **régulation de charge**.

2.8.3 La communication et la synchronisation

Le système p4 supporte un ensemble de primitives de communication entre processus : **send**, **receive**, et **diffusion**. L'opération **send** peut être **bloquante** ou **non bloquante**. Dans le cas d'un réseau hétérogène, le système attache à la communication une fonction de **codage/décodage** compatible avec le protocole XDR (SUN) pour assurer les conversions des données nécessaires en environnement hétérogène. Les procédures suivantes sont utilisées pour envoyer un message :

```
p4_send (type, to, msg, len)
p4_sendr (type, to, msg, len)
p4_sendx (type, to, msg, len, datatype)
p4_sendrx (type, to, msg, len, datatype)
```

Le paramètre **type** est un entier, choisi par l'utilisateur, qui définit un type de message. L'argument **to** identifie le processus à qui le message est envoyé et **msg** est l'adresse du message dont la taille est spécifiée par l'argument **len**. Les primitives avec un **r** utilisent le principe du rendez-vous : le processus reste bloqué jusqu'au moment où le récepteur reçoit le message. Le **x** dans la primitive indique la présence en paramètre d'une procédure (datatype) capable de coder le message dans une forme standard (protocole xdr). La primitive **p4_recv (rec_type, req_from, msg, len_recv)** permet de recevoir sélectivement un message. L'argument **rec_type** indique le type de message acceptable, (-1 indique type quelconque) et **req_from**

spécifie le processus (-1 indique processus quelconque) **msg** est le message reçu et **len_recv** contient sa taille.

Le système p4 offre un opérateur de diffusion. La primitive **p4_broadcast (type, data, data_len, data_type)** effectue une diffusion **globale** d'un message à tous les processus lourds décrits dans le fichier **procgroup**.

Pour la synchronisation de processus lourds sur différentes machines, le système offre la primitive **barrière**. Cette primitive permet aux processus de se synchroniser avant de continuer leur action. Les processus participants à une **barrière** continuent seulement après que le dernier soit arrivé à ce point (exécuter l'opération barrière). Les processus participants à une **barrière** sont tous les processus spécifiés dans le fichier **procgroup**. La portée d'une **barrière** comme d'une **diffusion** est globale.

Avec p4, on peut écrire un RPC en utilisant une combinaison des opérations **send** et **receive**. Le processus client peut faire un appel par :

```
p4_send(serveur, parametres, client) ;
p4_rec  (client, reponse) ;
```

Le paramètre **serveur** identifie le processus serveur, **paramètres** contient les informations nécessaires pour l'exécution de la procédure et **client** contient l'identification du demandeur du service. Après avoir envoyé la demande, le client est suspendu en attendant la réponse (opération receive). Le serveur possède une structure du type :

```
for(;;) {
   p4_rec (clients, parametres, client) ;
     -
     -
     -
   p4_send (client, reponse) ;
   }
```

Le processus serveur reçoit un appel avec une opération **p4_rec** où le paramètre **clients** spécifie un processus quelconque et **client** contient l'identification du processus demandeur. Après avoir exécuté l'action indiquée par les paramètres, le serveur envoie la réponse du RPC par une opération **p4_send**. Si le serveur doit synchroniser plusieurs requêtes entre elles, le seul moyen est disposer d'un processus léger pour recevoir systématiquement ces requêtes et créant des processus légers pour les traiter. Ceux-ci se synchronisent alors par l'intermédiaire de moniteurs.

2.8.4 L'environnement de programmation

Le système ne possède pas d'outils spécialisés pour la **production de programmes**, pour l'**aide à la prévision de performance**, pour la **correction** ni pour le **placement de programmes**. C'est uniquement une bibliothèque parallèle avec des mécanismes pour la communication, synchronisation et création de processus (machines UMA/NUMA). Par contre, il contient un ensemble de fonctions pour aider à la **à la mise au point** de programme. Ces fonctions permettent de tracer et enregistrer les événements de **création de processus**, **envoi de messages**, **réception de messages**, etc. Ces traces peuvent être analysés par des outils "post mortem" offerts par le système.

L'**environnement d'exploitation** utilisé par p4 est celui offert par le système Unix de la machine hôte. Ainsi, les opérations d'entrée/sortie sur l'écran et le clavier et les accès aux fichiers sont réalisés par les primitives Unix.

Le **placement statique** est fixé au lancement du programme. L'utilisateur crée un fichier de définition des processus puis, ces processus sont **placés** sur les processeurs en utilisant les mécanismes Unix (fork, rsh).

2.8.5 L'environnement d'exécution

p4 est un ensemble de fonctions et de macros qui forment une bibliothèque. Celle-ci, installée sur des machines NORMA ou UMA/NUMA, forme un **environnement de programmation parallèle**. p4 est implanté en utilisant les fonctionnalités du système Unix et les services réseau Unix TCP/IP. Il utilise la primitive Unix **fork** pour créer des processus sur le même processeur (localement). Pour créer des processus à distance, il utilise un **serveur spécial** ou la commande **rsh**. Le **serveur spécial** est un programme installé sur la machine distante, qui s'exécute en arrière plan et qui a la seule fonction de créer des processus. Si la création avec le serveur ne marche pas (serveur n'est pas installée ou il existe erreurs de connexion), elle est réalisée avec **rsh**.

2.9 Conclusion

A travers ces quatre études de cas nous avons vu trois modèles de programmation parallèle :

1. Le modèle de **processus communicants** : le système p4 est un modèle de processus communicants. Outre différentes formes de communication bi-point, il offre un mécanisme de diffusion et de barrière portant sur l'ensemble de processeurs. Aucune notion de diffusion ou barrière partielle (notion de groupe) n'existe. Le système supporte deux types de processus : lourds et légers. On ne peut démarrer à distance que des processus lourds (**création statique**) décrits dans un fichier de configuration et les processus légers (**création dynamique, locale**) sont gérés exclusivement par l'utilisateur.
   On notera que les **processus légers** sont implantés d'une manière lourde par les processus sur le système Unix.

2. Le modèle **client/serveur** : dans le modèle **client/serveur**, les processus sont partagés en clients et serveurs. Les clients demandent des services aux serveurs via un mécanisme d'appel de procédure : le RPC.

   Le langage Concurrent C suit ce modèle **client/serveur**. Le mécanisme proposé RPC (**transaction**) présente quelques variantes. Les processus sont définis **explicitement** et de manière **statique**. La synchronisation entre clients ne peut être assurée que par des serveurs programmés dans ce but et qui constituent un goulot d'étranglement potentiel.

   Le modèle de programme Orca est aussi le modèle **client/serveur**. C'est un nouveau langage adapté au parallélisme où les serveurs sont remplacés par des objets (de type moniteur). L'originalité provient de la réplication totale de ces objets, favorisant la parallélisation des accès, mais nécessitant un mécanisme de maintien de la cohérence. Le mécanisme propose est basé sur la diffusion atomique des modifications. La création des processus et objets est dynamique.

3. Le modèle à **mémoire partagée** : le modèle de programme Linda est un modèle à **mémoire partagée associative**. Linda propose deux principales abstractions :

tuples et **espace de tuples.** Un tuple est un couple **<clé, valeur>.** L'espace de tuples contient l'ensemble des tuples existants. Les processus communiquent et se synchronisent explicitement avec les opérations **in, out** et **read** sur les tuples. Un espace de tuples implanté de façon centralisée est un goulot d'étranglement. Dans un espace de tuples distribué sur des noeuds d'une machine NORMA, le problème principal est le maintien de la cohérence des données.

Un point commun de tous ces systèmes est de ne pas permettre la **migration** et la **régulation de charge.** Les **environnements de programmation** restent frustes. Nous présentons dans les tableaux suivants un résumé des caractéristiques principales des différents systèmes étudiés.

système	communication	synchronisation
Concurrent C	RPC	Serveur
Orca	RPC Diffusion globale	Serveur (objet)
Linda	Emission/réception explicites	Exclusion mutuelle
p4	canal virtuel Fifo, diffusion totale	moniteur (UMA) barrière (NORMA)

Table 2.4: Communication et synchronisation.

système	parallélisme	création	placement	régulation de charge	migration
Concurrent C	processus	statique	statique	non	non
Orca	processus	dynamique	dynamique	non	non
Linda	processus	dynamique	?	non	non
p4	processus lourds et légers	statique	statique	non	non

Table 2.5: Expression du parallélisme, placement et migration.

Environnement	Langage	modèle	machine
Concurrent C	extension de C	client/serveur	NUMA/NORMA
Orca	nouveau langage	client/serveur	NORMA
Linda	bibliothèque	mémoire associative partagée	NORMA
p4	bibliothèque	processus communicants/ mémoire partagée	NORMA UMA/NUMA

Table 2.6: Modèles de programmes et machines cible.

Nous allons présenter dans le chapitre suivant le modèle de programme que nous avons conçu. Il est destiné aux machines NORMA et est basé sur **l'échange de messages**. Il offre aussi une forme restreinte de **mémoire partagée**. Notre modèle est voisin de **p4**, mais la **communication** est **indirecte** avec le concept de **porte**, voisin de celui de boite à lettres. La **mémoire partagée** se présente comme un ensemble de portes particulières et ressemble à un espace de tuples dégénérée. La synchronisation est basée sur la notion de **barrière**. La **communication et la synchronisation** peuvent concerner plusieurs processus via une forme restreinte du concept de **groupe**. Enfin, **processus, portes** et **barrières** peuvent être créés **dynamiquement** sur un site quelconque spécifié par le processus père (pas de placement automatique).

Chapitre 3

Un modèle de programme parallèle

3.1 Introduction

Dans ce chapitre nous présentons le modèle de programme parallèle que nous avons conçu. Ce modèle de programme est accessible à travers du langage de programmation C//. Ce langage est une extension de la norme ANSI C qui permet la programmation concurrente d'une machine UMA. Le langage a été initialement développé comme outil pour une implantation parallèle de Prolog[Bri90a], [Bri90b],[Bri91], [Gey91], [Fav92]. Les concepts et mécanismes offerts dans ce langage sont les **threads**, les **sémaphores** et un mécanisme de **communication/-synchronisation** à la CSP (canaux, gardes). On peut trouver dans [Fav89] une description complète de ce langage.
Ce langage a été étendu par un ensemble de concepts et opérateurs parallèles faisant de C// un langage de programmation pour machine NORMA. Le concept de processus lourd (tâche) permet d'encapsuler les threads de C// partageant des données. Différents types et opérateurs permettent une programmation en terme de :

- création dynamique de tâches et processus,
- partage de données,
- appel de procédure à distance.

Ce modèle est une abstraction d'un modèle NORMA où les noeuds sont des machines UMA. En effet, une tâche avec ses processus que se partagent de la mémoire représente une machine virtuelle UMA ; des processus communicants qui ne partagent pas de mémoire, forment une abstraction d'une machine NORMA.

3.2 Expression du parallélisme

Le parallélisme est exprimé par l'intermédiaire de deux entités : **tâches** et **processus**.

Les tâches : une tâche est l'unité de ressources. Elle est définie par une image mémoire chargeable à placer dans la mémoire d'un processeur (UMA). Cette image est conforme à un modèle standard (segment de code, segment de données initialisées ou non, segment de piles des processus et segment tas). Une telle image peut être fabriquée à partir d'un programme C ou C//, et est en fait, un modèle de tâche. Le chargement d'une image mémoire de tâche précède le lancement de la tâche. Celui-ci correspond au lancement de son processus initial. Cette image mémoire peut être **placée** sur un, plusieurs, ou tous les processeurs du réseau.

Les processus : les processus sont les entités actives du système. Une tâche est le siège d'exécution d'un ou plusieurs processus qui se partagent la mémoire de la tâche. Les processus au sein d'une tâche se synchronisent par l'intermédiaire des sémaphores et peuvent communiquer par la mémoire commune ou par échange de messages. L'échange de messages utilise les canaux du langage C//. Un processus est créé **dynamiquement** soit localement à une tâche, soit à distance sur une autre tâche. Les processus de tâches différentes communiquent à l'aide du mécanisme de **portes**, que l'on présentera en 3.3, et se synchronisent par l'intermédiaire des **barrières**, qui seront détaillées en 3.4. La **création de processus à distance** est une innovation par rapport aux modèles **lourds/légers** classiques. Le processus créé à distance ne peut toutefois qu'exécuter une procédure locale à la tâche siège de son exécution.

Le parallélisme est **explicitement** déclaré par le programmeur. Un **programme parallèle** est un ensemble de **processus communicants**. Le processus initial d'une tâche

initiale commence l'exécution du programme parallèle. Une tâche, par l'intermédiaire de l'un de ses processus, peut créer d'autres tâches ainsi que d'autres processus, sur le même processeur ou sur un autre processeur.
Dans notre modèle, les processus ne sont pas anonymes, et tout le contrôle de l'exécution parallèle s'appuie sur leur identification.
Ainsi, bien que le système n'intègre ni la **migration** de tâches ni celle des processus, le **placement dynamique** de tâches et de processus permet de prendre en compte les problèmes d'équilibrage de charge des processeurs.

3.2.1 Tâches

Un processus d'une tâche peut **placer** une tâche ou **lancer** un processus **localement** ou sur un **autre processeur**. Un **processeur** est identifié par un simple numéro au sein d'une machine composé de processeurs homogènes. Deux opérateurs permettent de connaître l'identité du processeur courant et le nombre de processeurs disponibles dans le système :

```
int = MY_PROCESSOR_NUMBER () ;
```

retourne l'identification du processeur courant, et

```
int = PROCESSOR_NUMBER_MAX () ;
```

retourne le nombre de processeurs de la machine. Le nombre de processeurs est compris entre 0 et PROC_NUMBER_MAX (constante du système).

Un processus d'une tâche **place** une tâche sur un processeur par :

```
int i ;
i =  EXECTASK (proc_id, path, tid, nb-arg, taille1, arg1,
               taille2, arg2, ...,taillen, argn) ;
```

L'opération rend un code d'erreur entier. Les paramètres ont la signification suivante :

- **proc_id** : processeur où doit être **placée** la tâche,
- **path** : identificateur du modèle de la tâche, C'est un nom de fichier qui contient le programme (C, C//, ...) à exécuter,
- **tid** : après l'exécution d'EXECTASK, **tid** contient l'identificateur de la tâche créée.

La tâche démarre son exécution par son processus initial. Lorsqu'une tâche est créée, un bloc d'arguments peut lui être passé en paramètres (message initial) avec, par exemple, les identifications de portes de communication et de barrières. Dans EXECTASK, l'argument **nb-arg** spécifique le nombre d'arguments de ce bloc, et chaque argument est caractérisé par un couple **< taille, adresse de la valeur >.**

La structure générale d'une tâche est la suivante :

```
TASK_NAME identificateur
#
#inclusion de fichiers
#
definitions de constantes
definitions de variables globales
#
definitions des processus
#

BEGIN_TASK (arguments)  --bloc initial de donnees--
 {
   definitions des variables locales
   code d'initialisation de la tache
   EXECTASK (parametres) --creation de taches localement ou a distance--
   EXECTHREAD(parametres) --creation de processus localement ou a distance--
   LISTEN (liste-processus) --liste de processus creables--
   ALLWAIT(liste-id) --attente de la fin des processus dans liste-id--
   ANYWAIT(liste-id) --attente de la fin d'un processus dans liste-id--
   TERMINATE(liste-id) --termine les processus liste-id--
   ABORT() --termine le programme parallele--
END_TASK
```

Une tâche est un programme C// qui obéit aux règles syntaxiques de ce langage. Tous les opérateurs parallèles sont des procédures de la bibliothèque parallèle. Une tâche possède une partie **définition** et une partie **commande**. Dans la partie **définition** les inclusions de fichiers **en-têtes**, les définitions des **constantes et variables globales** sont réalisées, de la même manière que dans un programme C classique. Dans cette partie sont aussi définis les **processus** composants d'une tâche. La partie **commande** est le **processus initial** de la tâche et commence par BEGIN_TASK et termine par END_TASK. Il peut, facultativement recevoir des arguments (message initial). Il peut effectuer des initialisations, des créations et des placements de tâches, de portes et de barrières, déclarer la liste de processus exportés de la tâche,

etc. On doit remarquer que l'utilisation des opérateurs parallèles est permise à tous les processus de toutes les tâches, ce n'est pas une prérogative du seul processus initial de la tâche initiale.

3.2.2 Processus

Un processus est défini par :

```
THREAD  thread_name (parametres)
   BEGIN_THREAD
       declarations des variables locales

       instructions C//

   END_THREAD
```

BEGIN_THREAD et END_THREAD indiquent, respectivement, le début et la fin du code d'un processus. Un processus est équivalent à une fonction C, qui ne retourne pas de valeur. Les **paramètres formels** qui sont déclarés peuvent être des objets de notre environnement parallèle (portes, barrières, ..., etc.).

La primitive LISTEN (nb-proc, "p1", "p2", ..., "pn") permet au processus initial d'une tâche de déclarer la liste des types de processus qui peuvent être créés dynamiquement par des processus d'autres tâches. L'argument **nb-proc** en indique le nombre des procédures exportées et **p1**, **p2**, ..., **pn** leurs identificateurs (chaîne du caractères). Nous appelons un élément de cette liste un **point d'entrée de la tâche**. Elle joue aussi un rôle de **synchronisation**. En effet, une tâche termine quand son code d'initialisation termine (END_TASK) et que tous les processus actifs terminent ; l'opérateur LISTEN exprime alors le fait que la tâche (son processus initial) se met en attente de demande de création de processus. Le processus initial qui l'exécute est bloqué. Il ne redémarre plus et ne sortira de cet état uniquement que par une opération explicite de terminaison, TERMINATE, qui sera détaillée par la suite. L'argument **nb-proc** en indique le nombre des procédures exportées et **p1**, **p2**, ..., **pn** contiennent leurs identificateurs (chaîne du caractères). La primitive :

```
int i ;
i = EXECTHREAD (tache-id ,thread-name, tleger-id, nb-arg,
                taille1, arg1, taille2, arg2, ..., taillen, argn) ;
```

est utilisée pour créer un **processus localement** ou **à distance**. Cette opération retourne un entier qui contient le **code d'erreur** de la primitive. Les arguments signifient :

- **tache-id** : identificateur de la tâche qui contient le code du processus et qui encapsule le site de la tâche,
- **tleger-nom** : point d'entrée (chaîne de caractères),
- **thread-name** : identificateur unique du processus créé par l'opération (rendu en résultat).

Les arguments suivants sont les paramètres du processus créé, de la même manière que pour la création d'une tâche.

Les autres opérateurs de notre environnement parallèle (ALLWAIT, ANYWAIT, TERMINATE et ABORT seront présentés dans la section suivante.

3.2.3 Contrôle de l'exécution parallèle

Une tâche termine quand son processus initial termine ; un processus termine quand sa procédure termine. La fin d'une tâche, de la même manière que la fin d'un processus, n'occasionne pas la fin de ses tâches et processus fils. Il est à la charge de l'application de gérer la terminaison des entités qui ne sont plus nécessaires au programme. Le contrôle de l'exécution parallèle présente deux aspects :

- attente de terminaison des **tâches** et **processus** lancés,
- forcer la **terminaison** des processus cycliques (serveurs).

Le premier contrôle s'appuie sur la relation de filiation. Cette relation est du type **Maître/esclave**. Le processus créateur peut attendre sélectivement des tâches et processus fils. Cette forme offre deux primitives de contrôle :

1. ALLWAIT (liste-id) : le processus qui l'exécute se bloque jusqu'à ce que tous les processus spécifiés dans **list-id** terminent. L'argument **liste-id** contient une liste d'identificateurs de processus. Les listes de processus seront détaillées en 3.5.
2. ANYWAIT (liste-id) : cette primitive offre la possibilité d'une attente alternative. Lorsqu'un processus de l'ensemble décrit par **liste-id** termine, le processus qui a exécuté l'opération ANYWAIT est réveillé.

Le second contrôle concerne le problème de terminaison des tâches exportant des points d'entrée et qui sont en attente de création de processus. Le processus initial de ces tâches ne doit pas terminer tant qu'il existe des processus exportés en cours d'exécution ou qu'il nécessaire de créer des processus à distance. S'il est possible pour un serveur de décider de la terminaison des processus créés, il n'est pas possible de savoir si des créations de processus seront nécessaires dans le futur. Il ne peut pas donc décider de sa terminaison. C'est la terminaison du calcul qui doit déterminer la terminaison des serveurs. Le problème est alors traité en deux temps :

1. **empêcher le serveur de terminer** : la procédure LISTEN, outre la publication des **points d'entrée**, **suspend** le serveur.

2. **informer une tâche fille de sa terminaison** : la primitive TERMINATE permet au processus père d'informer une tâche de sa terminaison. L'effet d'un TERMINATE sur une tâche serveur est le suivant :

   - annule la publication des points d'entrée,
   - redémarre le processus initial de la tâche qui exécute alors le code de terminaison.

La primitive TERMINATE (liste-id) permet à l'utilisateur d'arrêter un, plusieurs, ou tous les processus au moment où l'action faite par un processus qui appartient à l'ensemble est terminée. L'argument **liste-id** contient la liste d'identificateurs des processus qui doivent terminer.
Soit un processus ayant exécuté la séquence d'opérations :

```
i1 = EXECTASK (t0) ;
i1 = EXECTASK (t1) ;
i2 = EXECTHREAD (t0, "th0", &p0) ;
i2 = EXECTHREAD (t1, "th1", &p1) ;
ALLWAIT(liste-id);  /* liste-id contient p0 et p1 */
TERMINATE(liste1-id); /* liste1-id contient t0 et  t1 */
---
"actions quelconques"
```

Si on suppose que **liste-id** contient les identificateurs des processus **p0** et **p1**, et que **liste1-id** contient les identificateurs des tâches **t0** et **t1**, le processus qui a exécuté cette séquence sera bloqué jusqu'à ce que les processus **p0** et **p1** terminent. A ce

moment, il reprend son exécution, termine les tâches **t0** et **t1** par TERMINATE, puis exécute les actions quelconques suivantes.

Dans notre système, tout le contrôle **d'attente de terminaison** et de **terminaison** doit être explicite dans l'application parallèle. Pour le système, il est impossible de savoir si les processus bloqués sont des **serveurs** (tâches bloquées par l'opération LISTEN) ou des processus en interblocage. La primitive ABORT force la terminaison des tâches et processus composants d'un calcul. Elle suspend l'exécution du programme parallèle.

3.3 La communication

Les processus communiquent par l'intermédiaire de **portes**. Une **porte** est gérée par le système et n'est pas associée au processus qui l'a créé. Une porte possède une identification unique et les messages sont émis et reçus sur une porte. La **création** d'une porte est faite **explicitement** sur un noeud localement ou à distance. Cette liberté d'utilisation et de placement implique de maîtriser les coûts relatifs de communication liés aux dépôts et retraits dans une porte.

La **réception** d'un message sur une porte n'est restreinte ni au processus créateur, ni aux processus du site porteur de la porte. Ainsi, si un processus exécute une opération **receive** sur une porte qui n'a pas été définie localement, la requête sera envoyée sur le site qui contient la porte et le retrait du message sera effectué.

Les tâches créent des portes, les reçoivent et les transmettent en paramètres lors de la création d'autres tâches (héritage) ou dynamiquement par des messages. Il existe deux types de portes : **Data et Message** . Les portes **Data** contiennent des données partagées entre processus. Une porte **Data** possède une seule structure de données associée que l'on peut lire ou écrire. Elles sont des portes FIFO et offrent une forme restreinte de **mémoire partagée**. Les portes **Message** sont utilisées pour l'échange de messages entre processus. Ces portes sont FIFO. Elles effectuent une **fusion** des flots de messages issus des différents processeurs émetteurs de façon à préserver l'ordre des émissions de chaque processus.

3.3.1 Les opérations sur les portes

Les portes du types **Message** et **Data** sont créées sur un processeur par :

```
int r0, r1 ;
r0 = PORT_MESSAGE_CREATE (processeur, &porte_message) ;
r1 = PORT_DATA_CREATE (processeur, &porte_data) ;
```

Après l'exécution de ces primitives, **porte_message** et **porte_data** contiennent les identificateurs uniques des portes créées. Le résultat des opérations est un code d'erreur.

Pour détruire une porte, les processus exécutent une opération DESTROY avec un paramètre identifiant la porte. La primitive :

```
PORT_MESSAGE_DESTROY(port_id) ;
```

élimine une porte du type Message et

```
PORT_DATA_DESTROY(port_id) ;
```

élimine une porte du type **Data**. Pour envoyer un message sur une porte du type **Message**, nous avons défini les primitives :

```
SEND (porte , &m , sizeof(m)) ;
SENDB (porte , &m , sizeof(m)) ;
```

La différence entre ces deux primitives est que le processus qui exécute **SEND** continue son exécution (**asynchrone**) et le processus qui exécute **SENDB** est bloqué (**synchrone**). Le paramètre **porte** identifie la porte destinatrice et les paramètres suivants explicitent l'adresse et la taille des données à transférer.
Pour recevoir un message, les processus utilisent les procédures :

```
r0 = RECV (porte , &m , sizeof(m)) ;
r1 = RECVNB (porte , &m , sizeof(m)) ;
```

Le paramètre **porte** identifie la porte réceptrice et les paramètres suivants explicitent l'adresse où le message doit être stocké et sa taille. La primitive **RECV** est bloquante. S'il n'existe pas de messages (file vide), le processus qui l'exécute est bloqué. La primitive RECVNB est non bloquante, elle retourne **FALSE** si la porte est vide.

Pour les portes de type **Data**, nous avons défini des opérations qui permettent la manipulation des données et la synchronisation. L'opération

```
GET_DATA (port_id, &data, sizeof(data)) ;
```

permet de récupérer la structure de données associée à une porte du type **Data**. Le premier paramètre spécifie la porte (locale ou distante), le deuxième l'adresse de mémoire pour stocker les données et le troisième spécifie sa taille. L'opération

```
PUT_DATA (port_id, &data, sizeof(data)) ;
```

est utilisée pour déposer des données dans une porte du type **Data**. Le premier paramètre spécifie la porte (locale ou distante) et les deux derniers l'adresse et la taille de la structure de données qui sera associée à la porte. Tous les processus qui connaissent la porte ont le droit d'exécuter cette opération. Les primitives GET_AND_LOCK et PUT_AND_UNLOCK sont utilisées pour bloquer et débloquer l'accès à une structure de données qui appartient à une porte du type **Data**, de manière à permettre une actualisation atomique. GET_AND_LOCK récupère les données associées à la porte et la bloque, et PUT_AND_UNLOCK associe des données et libère la porte. Une manière classique d'utilisation est le calcul atomique d'une nouvelle valeur en fonction de l'ancienne :

```
GET_AND_LOCK (port_id, &data, sizeof(data)) ;  | lecture et
   "calcul de la nouvelle valeur"              | modification
PUT_AND_UNLOCK (port_id, &data, sizeof(data)) ;| exclusive
```

Après un GET_AND_UNLOCK, tous les accès sont bloqués jusqu'au PUT_AND_UNLOCK.

3.4 La synchronisation

Trois mécanismes de synchronisation sont utilisables dans le système : les **sémaphores**, les primitives GET_AND_LOCK/PUT_AND_UNLOCK sur les portes **Data** et les **barrières**. Les **sémaphores** sont utilisables uniquement au sein d'une tâche par les processus d'une tâche. L'utilisation de la primitive GET_AND_LOCK permet de récupérer une structure de données et bloquer la porte associée. La primitive PUT_AND_UNLOCK fait l'**écriture** d'une valeur et libère l'accès à la porte.

Une **barrière** est définie comme un rendez-vous d'un ensemble donné de processus. Cet ensemble peut être définit par la liste des processus de l'ensemble, soit par la cardinalité de l'ensemble. Nous avons choisi cette solution. A toute **barrière** est donc attachée un quorum de processus. Une barrière est créée sur un processeur par :

```
BARRIER_CREATE (processeur, &barriere-id, quorum) ;
```

Le paramètre **processeur** spécifie le processeur où la barrière sera créée, **barriere-id** retourne l'identification de la barrière, et **quorum** spécifie le nombre de processus

qui participent initialement à la barrière. Un rendez-vous est demandé par un processus par :

```
BARRIER (barriere-id, quorum) ;
```

Cette primitive bloque le processus sur la barrière **barriere-id** jusqu'à ce que le quorum soit atteint. **quorum** est le nouveau quorum proposé par le processus. Pour qu'un nouveau quorum soit accepté, tous les processus participant au rendez-vous doivent être d'accord. Dans le cas contraire, le rendez-vous est effectué mais termine avec un erreur. La primitive

```
BARRIER_DESTROY (barriere-id) ;
```

termine la barrière **barrière_id**. Tous les processus qui sont bloqués sur la barrière (s'il y en a), sont débloqués.

3.5 Les actions pluralistes

Il est nécessaire dans un programme parallèle de répliquer des actions à destination de plusieurs processus ou processeurs. Par exemple, installer la même tâche sur plusieurs processeurs, lancer un même processus sur plusieurs tâches ou diffuser un message sur plusieurs portes. Nous avons introduit les concepts de liste **liste de processeurs**, **liste de processus** et **liste de portes**.

1. **La liste de processeurs** permet de répliquer une entité du système (tâche, porte) sur un ensemble de processeurs. Elle est composée d'identificateurs de processeurs. Un ensemble d'opérateurs permet la manipulation de ces listes.

2. **La liste de processus** permet de répliquer une création de processus sur un ensemble de tâches. Elle est aussi utilisée pour spécifier une attente de terminaison d'un ensemble de processus (primitive ALLWAIT), ainsi que pour terminer un ensemble de processus (primitive TERMINATE). Une **liste de processus** contient des processus locaux ou distants.

3. **Une liste de portes** est composée par un ensemble arbitraire d'identificateurs de **portes**. L'objectif de ce concept est d'étendre les possibilités des opérations de communication :

- d'attendre l'arrivée d'une communication sur une des portes de la liste de portes (**attente multiple**),
- de **diffuser** un message (porte Message) ou une valeur (porte Data) vers une liste de portes.

Les portes d'une liste peuvent être locales ou distantes. Les listes peuvent contenir, soit des portes du type **Message**, soit des portes des type **Data**. La liste de portes du type **Message** permet de faire une **diffusion** d'un message sur un ensemble de portes, ou **attendre pour un message sur une porte** de l'ensemble de portes. La **liste de portes Data** permet la **réplication** d'une structure de données sur un ensemble de portes.

Les listes de **portes**, **processus** et **processeurs** sont des structures de données locales à un processus. Leur format est standardisé du fait de leur utilisation dans les opérateurs de l'environnement parallèle : elles sont manipulables par des opérateurs de création/destruction et d'insertion/retrait d'éléments. A la différence des portes, barrières, ...,etc., ce ne sont pas des objets partagés dont la cohérence doit être maintenue vis à vis des processus les utilisant. Ainsi, le notion de liste de processus ou de processeurs ne définit elle pas un groupe au sens des systèmes distribués[Tan92].

3.5.1 Les opérations sur les listes

Pour chaque type de liste, quatre opérateurs permettent sa manipulation. Les opérateurs des **listes de processeurs** sont :

```
LPROCESSOR_CREATE(&lprocesseur)
LPROCESSOR_DESTROY(lprocesseur)
LPROCESSOR_INSERE(lprocesseur, processeur_id)
LPROCESSOR_DELETE(lprocesseur , processeur_id)
```

Pour la manipulation des **listes de processus**, les primitives sont les suivantes :

```
LPROCESSUS_CREATE(&lprocessus)
LPROCESSUS_DESTROY(lprocessus)
LPROCESSUS_INSERE(lprocessus, processus_id)
LPROCESSUS_DELETE(lprocessus, processus_id)
```

La primitive

```
LPROCESSOR_CREATE(&lprocesseur) ;
```

crée une liste vide de processeurs. A la fin de l'opération, le paramètre **lprocesseur** contiendra l'identificateur de la liste créée. Pour éliminer une liste de processeurs identifiée par **lprocesseur** on utilise la primitive :

```
LPROCESSOR_DESTROY(lprocesseur) ;
```

Les primitives :

```
LPROCESSOR_INSERE(lprocesseur, processeur_id) ;
LPROCESSOR_DELETE(lprocesseur, processeur_id) ;
```

sont utilisées, respectivement, pour ajouter ou enlever un processeur d'une liste.

Les opérations sur les **listes de processus** sont similaires à celles sur les **listes de processeurs**. On peut ajouter que dans l'opération LPROCESSUS_DESTROY, la liste est détruite, pas les processus qui font partie de la liste.

Les opérations qui permettent de manipuler les listes de portes sont :

```
LPORT_MESSAGE_CREATE(&lporte)
LPORT_MESSAGE_DESTROY(lporte)
LPORT_MESSAGE_INSERE(lporte, porte-id)
LPORT_MESSAGE_DELETE(lporte, porte-id)
LPORT_DATA_CREATE(&lporte)
LPORT_DATA_DESTROY(lporte)
LPORT_DATA_INSERE(lporte, porte-id)
LPORT_DATA_DELETE(lporte, porte_d)
```

Pour créer une liste vide de portes du type Message on utilise la primitive :

```
LPORT_MESSAGE_CREATE(&lporte) ;
```

A la fin de l'exécution de l'opération le paramètre **lporte** contiendra l'identificateur unique de la liste créée. La primitive :

```
LPORT_MESSAGE_DESTROY(lporte) ;
```

élimine la liste de portes identifiée par **lporte**. Les portes qui appartiennent à la liste ne sont pas détruites. Pour ajouter et enlever une porte d'une liste, on utilise les primitives :

```
LPORT_MESSAGE_INSERE(lporte, porte-id) ;
LPORT_MESSAGE_DELETE(lporte, porte-id) ;
```

Les opérations qui contiennent le mot "DATA" dans leur nom sont utilisées pour faire des opérations sur les listes de portes du type DATA et sont similaires à celles des listes de portes du type **Message**.

3.5.2 Réplication de tâches

Avec le concept de **liste de processeurs**, nous pouvons revoir les opérations de création de tâches et processus. Comme nous avons présenté en 3.2.1, le premier paramètre d'une opération EXECTASK définit le processeur où la tâche doit être placée. Ce paramètre peut être une **liste de processeurs**, ce qui permet de placer une même tâche sur un ensemble de processeurs. Dans ce cas, il faut aussi utiliser une **liste de processus** pour contenir l'ensemble des identificateurs des tâches créées. De manière identique, on peut utiliser une liste de processeurs pour la création d'une porte (Data ou Message) sur un ensemble de processeurs.
Dans l'exemple suivant, on crée une **liste de processeur** et une **liste de processus**. La liste de processeurs est initialisée avec les identificateurs des processeurs (entiers), et ensuite on exécute une opération EXECTASK pour répliquer la tâche, en donnant comme paramètre **"site de création" la liste de processeurs**. La tâche sera répliquée sur tous les processeurs de la liste. A la fin de cette opération, le paramètre **lprocessus** contiendra les identificateurs des tâches créées.

```
LPROCESSOR_CREATE(&lprocesseur) ;
LPROCESSUS_CREATE(&lprocessus) ;
nproc = PROCESSOR_NUMBER_MAX () ;
for(p=0;p<=nproc;p++)
  LPROCESSOR_INSERE(lprocesseur, p) ;
EXECTASK (lprocesseur, "tache-nom", lprocessus, nb-arg,
          taille1, arg1, taille2, arg2, ...,taillen, argn) ;
```

3.5.3 Réplication de processus

La primitive EXECTHREAD est utilisée pour démarrer un processus exporté par une tâche. Le premier paramètre doit indiquer la tâche où le processus est défini. Si on donne l'identificateur d'une **liste de processus**, on répliquera la création d'un processus sur toutes les tâches de la liste. Par exemple :

```
LPROCESSUS_CREATE(&lprocessus-id) ;
EXECTHREAD (lprocessus ,"nom", lprocessus-id, nb-arg, taille1, arg1,
            taille2, arg2, ..., taillen, argn) ;
```

Dans cet exemple, on crée une autre **liste de processus**, **lprocessus-id**, pour contenir les identificateurs des nouveaux processus. L'ensemble des identificateurs des processus créés sera retourné dans la liste **lprocessus-id**.

Les listes de tâches et de processus sont aussi utilisées pour spécifier une at-

tente de terminaison de processus, dans les opérations ALLWAIT, ANYWAIT et pour terminer un ensemble de processus, dans l'opération TERMINATE (3.2.3).

3.5.4 Diffusion d'un message

Nous pouvons faire une DIFFUSION d'un **Message** sur une liste de portes du type **Message** avec les primitives SEND et SENDB. Pour cela, le premier paramètre est l'identificateur d'une liste de portes, et non d'une porte. Par exemple, l'opération

```
SENDB (lporte_id,&m , sizeof(m))
```

est utilisée pour la **diffusion** d'un **message** sur une liste de portes **Message**. Le message est envoyée sur toutes les portes qui appartiennent à la liste. Le processus qui exécute cette opération continue seulement après que tous les messages **aient été reçus** sur chaque porte de la liste. On peut aussi faire une **diffusion** sur une liste de ports **Message** avec l'opération

```
SEND (lporte_id ,&m , sizeof(m))
```

Le processus qui l'exécute continue après que le message soit **déposé** dans la file de messages de chaque porte de la liste (pas reçus par le processus récepteur sur chaque porte). La diffusion faite avec l'utilisation des opérations SEND et SENDB est simplement FIFO. On ne garantit donc pas que les portes réceptrices enregistrent les messages issus de plusieurs diffusions dans le même ordre.

3.5.5 Écriture multiple

Pour **répliquer** une structure de données sur une liste de portes du type **Data** on utilise :

```
PUT_DATA (lporte_id ,&m , sizeof(m))
```

Les données sont associées à chaque porte de la liste, et le processus continue son exécution seulement après la fin de l'opération. Cette opération est aussi simplement FIFO. Nous n'avons pas des opérations atomiques dans le système.

3.5.6 Attente multiple

Il est suivant nécessaire d'exprimer l'attente d'un message parmi d'autres. Une forme est d'attendre un message quelconque arrivant sur un ensemble de portes. Ceci s'exprime par :

```
RECV (lporte , &m , sizeof(m)) ;
```

Le paramètre **lporte** est une liste de portes. Le processus est bloqué jusqu'au moment où un message arrive sur une des porte.

3.6 L'organisation des programmes parallèles

On distingue diverses formes de programmes parallèles (non exclusives) :

1. L'application est constituée d'un graphe de tâches ne comportant que des relations de précédence entre elles **(parallélisme de contrôle)**.
2. L'application est organisée autour d'une structure de données que des phases de calcul actualisent en parallèle **(data parallélisme)**.
3. L'application est organisé comme un ensemble d'activités potentielles, se créant les unes les autres. Un ensemble de travailleurs équivalents (ou non) exécutent les activités en essayant d'équilibrer leur charge de travail **(maître/-travailleurs)**.
4. L'application est définie par un ensemble de spécialistes interconnectés en un réseau logique véhiculant les données nécessaires au calcul **(processus communicants)**.

3.6.1 Parallélisme de contrôle

Un tel parallélisme se trouve dans un calcul exprimé par un enchaînement d'étapes, mais où la relation de dépendance entre étapes ne définit qu'un ordre partiel. C'est donc généralement un parallélisme qu'on peut extraire d'un programme séquentiel en analysant les dépendances entre étapes du programme (appels de procédure, par exemple). On peut trouver une traduction triviale de ce schéma dans notre modèle. Le programme s'exprime par une tâche initiale qui crée les tâches correspondants à chaque étape dès que les données initiales sont connues. Elles sont transmises via des portes Data. Cette tâche attend la terminaison des sous tâches pour recevoir les résultats nécessaires aux lancement d'autres tâches. Chaque sous tâche ainsi lancée procède de façon identique, sauf qu'à leur fin elles renvoient le résultat calculé. Un schéma possible de telles tâches est le suivant :

```
TASK_NAME etape
```

```
BEGIN_TASK (porte-in, args)
{
    GET_DATA (porte-in, donnees) ;
    ...
    PORT_DATA_CREATE (Data-out[i]) ;    --une etape se coupe en sous etapes--
    PORT_DATA_CREATE (Data-in[i]) ;
    PUT_DATA (Data-in[i], arg[i]) ;
    EXECTASK (St[i], Data-in[i], Data-out[i]) ;
    ...
    ALLWAIT(St[1], St[2],...,  St[n]);   --attente de la fin des sous taches--
    GET_DATA (Data-out[i], res-partiel) ;
    PORT_DATA_DESTROY (Data-out[i]) ;
    PORT_DATA_DESTROY (Data-in[i]) ;
    ...                                  --terminaison--
    PUT_DATA (porte-out, resultat) ;
END_TASK
```

Dans ce schéma considéré, une tâche n'est exécutée qu'une seule fois. Si plusieurs exécutions consécutives sont nécessaires (cas d'une procédure), il nous faut éviter de payer le coût d'installation de la tâche (chargement) à chaque appel. Le schéma serait alors de partager chaque tâche en deux parties :

- une partie est responsable de l'installation de sous tâches,
- une partie effectuant le calcul dévolue à la tâche et appelant d'autres tâches.

Une tâche de ce type se présente avec :

- un code d'initialisation responsable de l'installation/terminaison des sous tâches,
- un code de calcul exporté comme point d'entrée.

```
TASK_NAME etape

THREAD calcul (Port-in, Port-out) --la partie calcul est un point d'entree--
    BEGIN_THREAD
       GET_DATA (porte-in, donnees) ;
       ...
       PORT_DATA_CREATE (Data-out[i]) ; -- invocation des sous calculs --
       PORT_DATA_CREATE (Data-in[i]) ;
       EXECTHREAD (T[i], calcul[i], Data-in[i], Data-out[i]) ;
```

```
    ...
    ALLWAIT(calcul[1], calcul[2], ..., calcul[n]);  --attente de la fin--
    ...                                             --des sous calculs--
    GET_DATA (Data-out[i], res-partiel) ;
    PORT_DATA_DESTROY (Data-in[i]) ;
    PORT_DATA_DESTROY (Data-out[i]) ;
    ...
    PUT_DATA (porte-out, res) ;
END_THREAD

BEGIN_TASK ()   -- partie installation --
{
    EXECTASK (T[i]) ; -- installer les taches necessaires --
    ...               -- aux sous calculs --
    LISTEN ("calcul") ;
    TERMINATE (T[i]) ;  -- terminer les taches appelees --
END_TASK
```

La tâche initiale est en fait le chargeur-placeur initial :

```
TASK_NAME init

BEGIN_TASK ()
{
    EXECTASK (T[0]) ; -- chargement de la  tache initiale --
    ...
    EXECTHREAD (T[0], calcul[0], Data-in[0], Data-out[0]) ; --calcul initial--
    ALLWAIT (calcul[0]) ; -- attente de la fin de calcul[0] --
    TERMINATE (T[0]) ;  -- terminer la tache initiale --
END_TASK
```

3.6.2 Parallélisme de données

Le parallélisme de données provient d'une adaptation parallèle d'un schéma séquentiel (itération) d'accès à un ensemble de données (tableau). Dès que deux calculs consécutifs portent sur des parties disjointes de l'ensemble, ces calculs peuvent être faits en parallèle. Par contre, une synchronisation est nécessaire du fait que ces calculs peuvent être précédes ou suivis par des calculs ne vérifiant pas cette indépendance d'accès aux données. L'exemple suivant est celui d'un calcul sur un tableau itérant jusqu'à ce que le tableau vérifie une propriété. Une étape d'itération se décompose en calculs élémentaires disjoints nécessitant la donnée du tableau à l'étape précédente et calculant une partie de la nouvelle valeur du

tableau. Le tableau est défini par un ensemble de portes Data qui contiennent chacune une partie du tableau. N processus vont exécuter en parallèle une itération calculant une partie à partir de toutes. La synchronisation des phases d'itération est assurée par une barrière.

```
==============================
*          Phases            *
==============================

TASK_NAME Phases -- tache enchainant des phases de calcul --


BEGIN_TASK (int ntache, t_porte_data porte0, t_porte_data porte1,
               t_porte_data porte2, t_barriere b)
 {
  t_global tlocal_data[3] ;
  t_porte_data porte[3] ;
  int i ;

    porte[0] = porte0 ;
    porte[1] = porte1 ;
    porte[2] = porte2 ;
    while ("condition donne"){
          -- recuperation de la valeur de l'etape i --
          for(i=0;i<3;i++;)
             GET_DATA (porte[i], &tlocal_data[i]) ;
          BARRIER(b,3);
          "calcul" (tlocal_data)
          -- production de la valeur de l'etape i+1 --
          PUT_DATA (porte[ntache], &tlocal_data[i]) ;
          BARRIER (b,3) ;
          }
END_TASK

TASK_NAME Program

BEGIN_TASK ()
{
  tache_id t[3] ;
  t_porte_data porte[3] ;
  t_barriere b ;
  t_lprocessus lp0 ;
```

```
  t_global global_data[3] ; -- une donnee initiale --
  t_global res[3] ; -- resultat final --
  char pathname [] = "Phases" ;
      my_proc = MY_PROCESSOR_NUMBER () ;
      BARRIER_CREATE (my_proc+1, b, 3) ;
      for(i=0;i<3;i++) {                --instalation des portes data--
         PORT_DATA_CREATE (my_proc+i, porte[i]) ;
         PUT_DATA (porte[i], &global_data[i]) ;
         }
      for(i=0;i<3;i++) {
         EXECTASK (my_proc+i, pathname, &t[i], 5, sizeof(int), i,
                     sizeof(t_porte_data), porte[0],
                     sizeof(t_porte_data), porte[1],
                     sizeof(t_porte_data), porte[2],
                     sizeof(t_barriere), b) ;
         LPROCESSUS_INSERE(lp0, t[i]) ;
         }
      ALLWAIT(lp0);
      BARRIER_DESTROY (b) ;
      for(i=0;i<3;i++) {
         GET_DATA (porte[i], res[i]) ; -- recuperation du resultat --
      ...
      for(i=0;i<3;i++) {
         PORT_DATA_DESTROY (porte[i]) ;
END_TASK
```

Dans cet exemple, trois porte du type **Data** et une **barrière** sont utilisées pour la synchronisation des différentes phases de calcul. Le programme parallèle est composé de quatre tâches. La tâche **initiale** est **placée** sur le processeur 0 par une fonction initiale de **placement**. Elle démarre et crée trois portes **Data** avec la primitive PORT_DATA_CREATE et une **barrière** avec la primitive BARRIER_CREATE, qui spécifie aussi que trois processus vont se synchroniser. Une porte est **placée** sur chaque processeur (**my_proc+1**, **my_proc+2** et **my_proc+3**) et la **barrière** est créée sur le processeur **my_proc+1**. Ensuite, avec l'opération PUT_PORT (porte[i], &global_data, sizeof(global_data)) les portes **Data** sont initialisées avec la valeur **global_data**. Puis, la tâche initiale lance trois tâches **phases**, et envoie comme paramètre l'indice de la tâche, les portes et la barrière. La tâche initiale fabrique une liste de processus qui contient les identificateurs des tâches, et se bloque en attendant la fin de tous les tâches filles,primitive ALLWAIT, qui possède comme

paramètre la liste qui contient les identificateurs des tâches.
Les tâches exécutent une séquence synchrone d'itération. Chaque itération d'une tâche est une phase de calcul. Dans la phase i, les trois tâches récupèrent les valeurs des trois portes. L'opération **barrière** assure que toutes les tâches prennent les mêmes valeurs. Chacune fait le calcul de la nouvelle valeur d'une porte. Ensuite, elles se synchronisent une autre fois pour permettre l'actualisation de toutes les portes, puis recommencent une nouvelle phase de calcul. Quand les tâches terminent, par une condition donnée, la tâche initiale est réveillée termine toutes les portes et la barrière, et termine.

3.6.3 Processus communicants

Dans ce schéma, l'application se décrit comme un ensemble de tâches interconnectées par des portes Message. Une tâche initiale assure la création des portes et l'installation des tâches. L'exemple suivant montre un réseau logique de type anneau où la tâche initiatrice initie le fonctionnement de l'anneau et le termine.

```
==============================
*          anneau            *
==============================

TASK_NAME stage-i

BEGIN_TASK (t_porte p-in, t_porte p-out)
{
  RECV (p-in, m, sizeof(message)) ;
  "calcul"
  SEND (p-out, m, sizeof(message)) ;
END_TASK

TASK_NAME initiatrice

BEGIN_TASK ()
{
  tache_id tid[6] ;
  t_porte p[6] ;
  int i, i0, i1, i2, i3, i4 ;
  my_proc = MY_PROCESSOR_NUMBER () ;
  for(i=0;i<6;i++)
     PORT_MESSAGE_CREATE (my_proc+(i+1mod6, &p[i]) ;
```

```
  i0 = EXECTASK (my_proc+1, "stage-i", tid[0], 2,
                 sizeof(porte-id), p[0],
                 sizeof(porte-id), p[1]) ;
  i1 = EXECTASK (my_proc+2, "stage-i", tid[1], 2,
                 sizeof(porte-id), p[1],
                 sizeof(porte-id), p[2]) ;
  i2 = EXECTASK (my_proc+3, "stage-i", tid[2], 2,
                 sizeof(porte-id), p[2],
                 sizeof(porte-id), p[3]) ;
  i3 = EXECTASK (my_proc+4, "stage-i", tid[3], 2,
                 sizeof(porte-id), p[3],
                 sizeof(porte-id), p[4]) ;
  i4 = EXECTASK (my_proc+5, "stage-i", tid[4], 3,
                 sizeof(porte-id), p[4],
                 sizeof(porte-id), p[5]) ;
  SEND (p[0], m  , sizeof(message)) ;
  RECV (p[5], m  , sizeof(message)) ; -- recuperer les resultats --
  ...
  for(i=0;i<6;i++)
     PORT_MESSAGE_DESTROY (&p[i]) ;
END_TASK
```

D'une manière générale, une application de type **processus communicants** se décrit comme une tâche initiale installant un réseau de tâches où chacune de ces tâches à son tour installe un sous réseau. Ainsi, dans l'exemple précédent la tâche initiatrice recevait les données à injecter dans l'anneau depuis des portes passées en paramètres et y produisant de même les résultats extraits de l'anneau. On aurait avantage à ce niveau à créer deux processus locaux responsables de l'injection/éjection de l'anneau.

3.6.4 Maître/travailleurs

Dans ce modèle le programme est découpé en un certain nombre de tâches capables d'exécuter tout ou partie des actions nécessaires au programme. Chaque action peut exiger un nombre variable d'autres actions concurrents qui ne sont connues qu'à l'exécution. Le problème consiste généralement à répartir les actions à exécuter de façon efficace sur les tâches capables de les faire (équilibrage de charge). Quand toutes les tâches sont équivalentes, on appelle un tel modèle **processor farm**. Dans notre environnement une action élémentaire correspond à

un point d'entrée et une tâche regroupe plusieurs points d'entrée. Si l'invocation d'un point d'entrée s'exprime aisément par un EXECTHREAD, c'est moins facile de sélectionner la tâche donnant la meilleur efficacité. Une possibilité d'implantation est d'interdire un processus d'appeler directement une tâche pour initier une action. Ils doivent transmettre cette requête à une tâche particulière, l'ordonnanceur, qui connaît en permanence l'état de charge des autres tâches. C'est l'ordonnanceur qui effectue les EXECTHREAD sur la tâche la "moins chargée". Dans ce cas, l'ordonnanceur est centralisé.

Une autre solution est que chaque thread lancé met à jour l'état de la tâche (porte Data par exemple). Cet état de charge pourrait être consulté pour un thread désirant initier une action et cherchant une tâche "sous chargé". Notre modèle de programmation n'offre donc aucune facilité pour régler le problème de la régulation de charge qui reste du domaine du programmeur et de la recherche d'une manière générale.

3.7 Comparaison avec les systèmes étudiés

Dans le chapitre précédent nous avons présenté les langages Concurrent C et Orca, Linda et le système p4. Les langages Concurrent C et Orca sont basés sur le mécanisme RPC. Ce mécanisme aussi peut être programmé en Linda et p4.

3.7.1 Le protocole RPC

Notre modèle supporte un ensemble d'opérations de communication entre processus, mais cet ensemble n'inclut pas le RPC. Nous allons montrer que le RPC peut être programmé dans notre modèle d'une manière simple grâce à la création de processus à distance. Le processus client peut faire un appel par :

```
RPC (serveur, operation, parametres, resultat)
 {
   t_porte  p ;
   message  m ;
   int i ;
   PORT_MESSAGE_CREATE (my_processor, &p)
   "emballage des parametres"
```

```
  m.porte_cli = "p" ;
  m.operation = "op" ;
  m.parametres = "parametres" ;
  i = EXECTHREAD("serveur", "op", &tid, 1, sizeof(m), m) ;
  RECV (p, &reponse, sizeof(reponse)) ;
  "deballage des parametres"
 }
```

Le paramètre op doit être un point d'entrée exporté par la tâche "**serveur**". Le "**point d'entrée** (processus qui sera créé) est définit par :

```
THREAD operation (message * m)
  BEGIN_THREAD
    "deballage des parametres"
        "action"
    SEND (m.porte_cli, resultats) ;

  END_THREAD
```

Le seul point important est de savoir comment le client peut connaître la table de serveurs. Ce problème est laissé au concepteur de l'application. On peut imaginer un **serveur de serveurs**, ou simplement une porte Data enregistrant les localisations des serveurs. Une tâche serveur lors de l'initialisation s'enregistre préalablement à l'exploitation des points d'entrée. Si l'on ignore ce problème, le concept de point d'entrée et la création de processus à distance est un mécanisme plus général et plus puissant que le simple RPC synchrone.

3.7.2 Espace de tuples

Avec notre modèle nous pouvons implanter une forme d'espace de tuples simplifié, distribué, en utilisant la porte **Data**. Les tuples :

- sont placés explicitement sur les noeuds,
- ne migrent pas,
- ne sont pas dupliqués.

L'espace de tuples est l'ensemble des portes **Data**, distribuées sur les processeurs de la machine.

3.7.3 Bilan

Le point fort de Linda est le mécanisme de mémoire associative constituée par l'espace de tuples. Le défaut est une implantation complexe et une sémantique de communication imprécise du point de vue de la synchronisation associée. Le problème de la gestion de processus doit être traité par le système hôte du modèle.

Dans le langage Orca, l'utilisateur ne traite pas directement la communication. Il exécute des opérations sur des objets, qui sont actualisés (quand nécessaire) par un mécanisme de **diffusion atomique**, réalisant ainsi une mémoire d'objets partagés. Ceci se paie par une implantation complexe reposant sur l'efficacité du réseau d'interconnexion véhiculant les diffusions.

Notre modèle est proche de celui de p4. Nous avons des tâches et processus qui matérialisent des machines NORMA et UMA, mais la communication dans notre modèle est faite de manière **indirecte**, par le mécanisme de **portes messages**. On peut **placer** des **portes**, des **tâches** et des **processus dynamiquement** sur un processeur, ce qui est impossible dans p4, mais possible en Orca. Le concept de **point d'entrée** et de création de processus à distance nous offre des fonctionnalités plus générales que le RPC simple de Concurrent C et est équivalent au **fork** de Orca. Le mécanisme de **porte Data** est une forme simple de mémoire partagée, bien inférieure à Orca ou à Linda, mais d'implantation très simple.

Notre modèle nécessite le placement de tous ces objets. En ce sens, notre système est homogène, car tous les objets (tâches, processus, portes et barrières) doivent être placés par le programmeur, alors qu'en Linda et Orca il pourrait être difficile de concilier les placements explicites de processus et les placements/migrations automatiques des objets mémoire.

En ce qui concerne Concurrent C, il n'offre pas d'avantages sur C//. Tous les deux sont restreints à une utilisation sur machine UMA. Concurrent C est basé sur un modèle client/serveur inspiré de ADA[New87], alors que C// est inspiré de CSP[Hoa78]. Cependant, C// permet la création dynamique de processus, contrairement à Concurrent C. Le tableau suivant résume les caractéristiques principales des différents modèles.

Aspects	Concurrent C	Orca	Linda	p4	Notre modèle
Processus dynamiques	Non	Oui	Oui	Non	Oui
Placement initial de processus	Non	Oui	?	Non	Oui
Placement initial de tâches	Oui	Non	?	Oui	Oui
Migration	Non	Non	Non	Non	Non
Communication	Non	Non	Oui	Oui	Oui
Barrière	Non	Non	Non	Oui	Oui
Diffusion	Non	Implicite dans les opérations qui modifient les objets	Avec l'opération read sur une tuple par les processus concernés	Oui	Avec SEND et SENDB sur les listes de portes
RPC	Avec les transactions	Avec les opérations sur les objets	Non	Non	Avec la création des processus à distance
Mémoire partagée	Non	Avec la duplication des objets	Avec l'espace de tuples	Non	Avec les portes Data

Table 3.1: Les principaux aspects des systèmes étudiés.

Chapitre 4

L'environnement d'exécution

4.1 Introduction

Dans le chapitre précédent, nous avons défini les concepts et les opérateurs de notre **modèle de programme parallèle**. Nous allons présenter la réalisation de cet environnement sur une machine parallèle NORMA à base de Transputers : Supernode.

4.1.1 Présentation de la machine parallèle

Le **Supernode** (figure 4.1) est une machine NORMA composée de processeurs Transputer. Dans sa version Tnode, elle peut avoir au maximum 32 processeurs de travail (16 dans la machine utilisée), destinés à l'exécution de programmes utilisateurs. Son réseau d'interconnexion est reconfigurable par deux commutateurs de liaisons (**"switchs RSRE et C004"**). Un processeur de contrôle (**contrôleur**) en assure la commande de la machine. Les quatre liens des processeurs de travail sont liés au commutateur RSRE. Il est possible ainsi de créer un graphe quelconque d'interconnexion de degré 4.

Le **contrôleur** est responsable du contrôle de la machine. Il commande la configuration du réseau d'interconnexion et assure le contrôle des processeurs de travail par un bus spécial pour le **débogage**, la **synchronisation** et **l'initialisation** des processeurs.

Le **Supernode** a été conçu pour être attaché à une machine frontale qui doit lui

fournir les ressources **d'entrée/sortie** et de **gestion de fichiers** qui lui font défaut. Le commutateur **C004** est plus spécialement destiné à l'aiguillage de certaines des liaisons de la machine vers l'extérieur. C'est par l'intermédiaire de cet interface que s'effectue la liaison avec la machine hôte pour le chargement initial du Tnode, ou l'accès aux serveurs (entrée/sortie, fichiers).

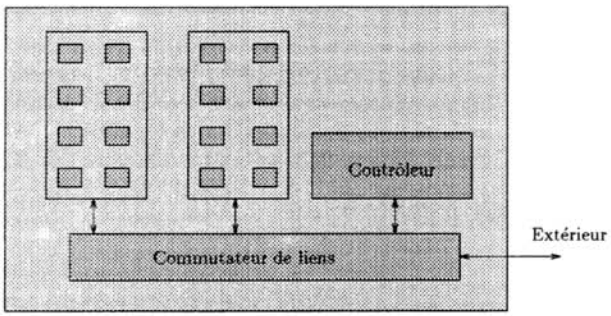

Figure 4.1: La machine Tnode à 16 Transputers.

4.1.2 Concepts et architecture

Comme nous l'avons indiqué dans le chapitre précédent, un environnement d'exécution pour machines NORMA offre des services de **gestion de tâches** (création et destruction de tâches locales ou distantes), de **communication** entre processus sur le même processeur ou sur des processeurs différents, de **synchronisation** et aussi de gestion de la **mémoire**. A ceci, on peut ajouter des services tels que la gestion des **fichiers** et la gestion des **entrées/sorties**.

1 Micro Noyau Parallèle

Un noyau de système parallèle peut être organisé de façon classique. Chaque processeur possède un noyau de système monolithique (figure 4.2) auquel s'ajoutent la gestion des communications et l'accès aux services "classiques" distants. L'ensemble des noyaux monolithiques coopérants forme un **noyau monolithique parallèle**.

Une autre solution plus récente consiste à suivre l'approche dite de **Micro Noyau**.

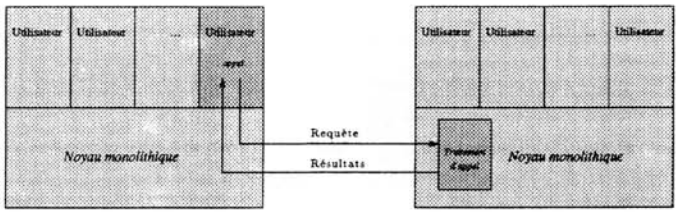

Figure 4.2: Organisation typique d'un noyau monolithique parallèle.

Un **Micro Noyau** est essentiellement un **noyau de communication entre processus**, de **gestion de processus** et de **gestion de la mémoire**. Il n'offre aucun des services classiques (fichiers, entrée/sortie). Ceux-ci sont assurés par des processus serveurs. Par exemple, l'ouverture d'un fichier se traduit par une requête du processus client vers le serveur **gestionnaire de fichiers**. Ce serveur exécute la requête et envoie la réponse au client. La figure 4.3 montre l'organisation d'un Micro Noyau. Les avantages apportées par un Micro Noyau sont la flexibilité et la modularité. Tous les ressources du système sont accessibles de la même manière, selon un protocole client-serveur. Pour ajouter un nouveau service, il suffit de rajouter un nouveau serveur. Un **Micro Noyau Parallèle** est donc composé par un ensemble de

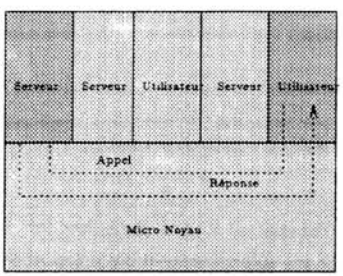

Figure 4.3: Structure d'un Micro Noyau.

Micro Noyaux locaux coopérants, un sur chaque noeud. La fonction de communication de chaque Micro Noyau est étendue de façon à permettre l'accès à des services offerts par des serveurs distants (figure 4.4).

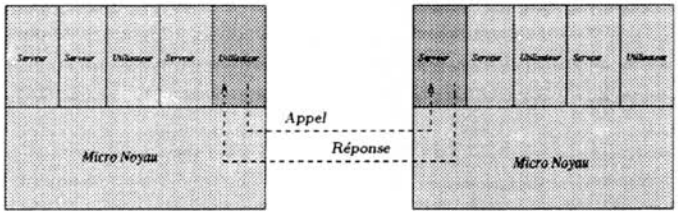

Figure 4.4: La communication entre client et serveur distant dans un Micro Noyau Parallèle.

2 Micro Noyau de Communication

Le point crucial d'un Micro Noyau Parallèle est le Micro Noyau de Communication qui permet la communication entre processus (sur un même processeur ou non) selon le protocole client-serveur choisi. Une organisation typique d'un **noyau de communication** est la structuration en couches OSI[Tan90]. L'organisation OSI classique dans ses couches basses est la suivante :

- couche 1 : physique
- couche 2 : liaison de données
- couche 3 : réseau
- couche 4 : transport
- couche 5 : session

Dans le cas de la machine Supernode, l'équivalent des couches 1 et 2 est assuré par le processeur Transputer. Il reste à réaliser les couches 3, 4 et 5. La couche 4 joue généralement un rôle d'adaptation d'une technologie système donnée et d'une technologie réseau qui doivent pouvoir évoluer indépendamment. Dans le cas d'une machine parallèle, il faut rechercher l'intégration la plus éfficace. C'est porquoi nous considérons que le noyau de communication est composé de deux parties seulement : la partie **protocole(s)** et la partie **acheminement de message(s)**.

La couche protocoles

À ce niveau sont traités les protocoles de communication entre processus et les synchronisations associées. Chaque protocole suppose un traitement particulier à l'expédition et à la réception. Nous pouvons envisager deux possibilités d'organisation de cette couche. Selon la première, chaque message est typé par un protocole et chaque type définit le traitement particulier du message correspondant au protocole. L'autre solution est d'avoir un seul protocole privilégié suffisamment puissant pour implanter sur lui les autres protocoles. Par exemple, un mécanisme RPC de base assure les seuls transferts de requête/réponse à destination des serveurs. Les autres protocoles seraient implantés par des serveurs de communication. Les figures suivantes montrent ces deux organisations.

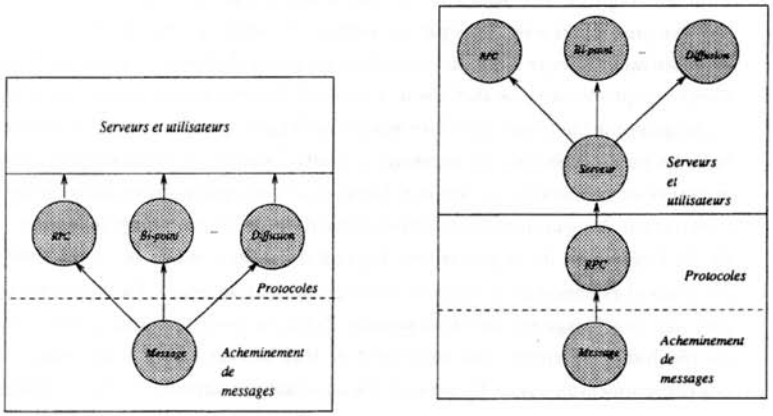

Figure 4.5: La couche protocole.

Présentation générale du protocole RPC

Dans une organisation d'un système ou d'un environnement d'exécution basée sur la philosophie de Micro Noyau Parallèle, les serveurs offrent des services aux utilisateurs (clients). L'utilisation d'un mécanisme d'appel RPC est bien adapté

à l'implantation du modèle client/serveur car il est simple, possède une interface précise et permet d'obtenir un comportement identique à l'appel de procédure système des noyaux traditionnels.

En effet, dans les organisations traditionnelles, les services sont assurées par des procédures appelées par les programmes clients. Ces procédures sont invoquées en utilisant des traits spécifiques (piles, appel de sous programme et déroutement). Les programmes clients ont connaissance des adresses de ces procédures, soit de façon conventionnelle (numéro de fonction du système) soit par édition de lien (bibliothèque).

Pour les organisations basées sur un Micro Noyau Parallèle, un service est assuré par des procédures exécutées par un serveur. Le schéma type de fonctionnement est le suivant : à toute fonction/procédure appelée à distance, il correspond chez le client une procédure dite **stub client**. L'objectif de cette procédure est de proposer au programme client une interface adapté au langage utilisé. Cette procédure emballe les paramètres dans un message, y ajoute l'identité de la procédure invoquée et transmet ce message au serveur. Celui-ci à la réception du message procède à l'extraction des paramètres et initie l'exécution de la procédure invoquée. A la fin de l'exécution de la procédure, l'opération inverse est faite : emballage des résultats et émission de la réponse vers le processus appelant. La procédure assurant cet interfaçage est dite **stub serveur**. Enfin, la procédure **stub client** récupère les résultats et termine. Les **stub client** et **stub serveur** assurent la transparence des communications pour les procédures appelante et appelée. Ce fonctionnement met en jeu cinq éléments :

- le processus client,
- le processus serveur,
- la procédure appelée à distance,
- la procédure stub client,
- la procédure stub serveur.

Une qualité importante de ce schéma est qu'il repose uniquement sur un modèle de processus communicants. Il fonctionne indépendamment de la localisation res-

pective des clients et du serveur. S'ils sont sur la même machine la communication se fera localement, sinon elle se fera via le réseau. La figure 4.6 montre un appel à un serveur local et la figure 4.7 présente l'échange de messages entre client et serveur distant. Dans une présentation de systèmes utilisant le RPC, Tay et

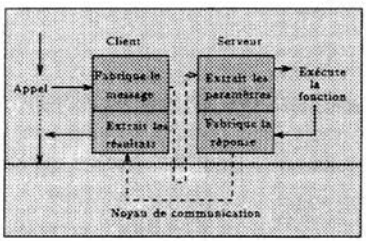

Figure 4.6: Echange de messages entre client et serveur local.

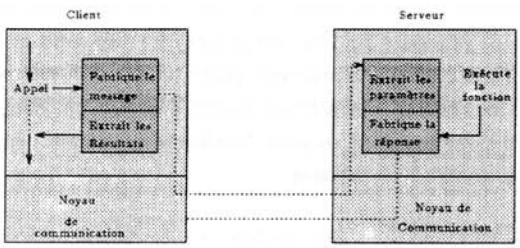

Figure 4.7: Echange de messages entre client et serveur distant.

Ananda[Tay90], [Ana92] distinguent RPC **synchrone** et RPC **asynchrone**.

Le modèle synchrone

Dans un appel RPC **synchrone**, le client est suspendu jusqu'au moment où la réponse arrive. Une critique de ce modèle est qu'il n'exploite pas les possibilités

de parallélisme existant dans les applications parallèles et distribuées. Le RPC **synchrone** est aussi inadéquat lorsqu'une opération de diffusion est nécessaire ou dans un calcul où un résultat est fourni incrémentalement par le serveur.

Le modèle asynchrone

Dans un RPC **asynchrone**, l'appel ne suspend pas le client. Les réponses, si elles existent, sont reçues quand elles deviennent nécessaires à la poursuite du calcul. Ainsi, le client continue l'exécution en parallèle avec le serveur. On trouve en [Sat87], [Lis88], [Wal90] et [Ana92] des solutions qui sont utilisées pour l'implantation du RPC **asynchrone** dans quelques systèmes.
Indépendamment du contrôle du parallélisme entre le client et le serveur, il peut y avoir un contrôle du parallélisme dans l'exécution des procédures de service. Les exécutions peuvent être exclusives les unes des autres. A l'autre extrême, on peut admettre autant d'exécutions concurrentes qu'il y a d'appels à un moment donné. Dans ce cas, pour chaque appel reçu par un serveur, celui-ci crée un processus pour l'exécuter. Les critiques que l'on peut faire à cette deuxième solution est que le nombre de processus peut saturer la mémoire disponible et créer un blocage. Par ailleurs, la création de processus peut être une opération lourde, dégradant l'efficacité du système. Une solution intermédiaire est que chaque serveur ait un nombre limité de processus esclaves. Le rôle du serveur est alors de distribuer les requêtes de service à ces esclaves.

Le modèle RPC comporte deux problèmes importants à résoudre : l'**interblocage** et l'**adressage des serveurs**.

Le problème d'interblocage

Il existent des situations où les serveurs ont besoin de communiquer. Par exemple, le serveur **gestionnaire de tâches** communique avec un **gestionnaire de tâches** d'un autre processeur pour créer processus distant. Dans un RPC **synchrone** un problème qui peut apparaître, dans la communication entre serveurs est l'**interblocage**.

Si on considère le cas où un serveur A envoie un message pour un serveur B, en demandant un service et que le serveur B envoie, en même temps, un message pour le serveur A, il apparaît un cas d'interblocage.

Deux politiques de traitement sont possibles. La première interdit purement et simplement qu'un serveur envoie des messages à un autre serveur. La deuxième, autorise les requêtes de serveur à serveur mais il faut alors se prémunir contre l'interblocage. Il suffit d'interdire la création d'un cycle dans le graphe de dépendance client/serveur. Une première méthode consiste à classer les serveurs en couches. Un serveur d'une couche donnée ne peut appeler que les serveurs d'une couche inférieure. Cette solution a été employée dans le système Minix[Tan87] La seconde méthode autorise une communication sans restriction mais il est toujours nécessaire d'éviter la création d'un cycle. La solution consiste à créer pour chaque requête un processus chargé de l'exécuter. Un serveur de ce type est donc constitué d'un processus maître à l'écoute des requêtes et créant les processus esclaves pour leur exécution. Un serveur ne peut donc engendrer de blocage tant qu'il peut créer un esclave. En dernier ressort, l'interblocage est introduit par un échec de création de processus, c'est à dire, un échec d'allocation de mémoire. C'est un problème grave dans le cadre d'un système d'exploitation multiusagers, car il est difficile de savoir à qui est due l'erreur. Dans le cas d'un noyau exécutif pour programme parallèle, cette erreur traduit simplement l'insuffisance de la mémoire d'une machine pour un algorithme parallèle donnée.

L'adressage des serveurs

Pour envoyer une requête le client doit connaître la localisation et l'identification locale correspondant à un serveur. C'est un problème classique des systèmes distribués. Les serveurs ont un nom qui permet de les désigner les uns par rapport aux autres. Les serveurs ont une localisation qui permet de les atteindre. Une localisation est une paire **<adresse processeur, nom local>**. Il y a diverses façons d'établir la correspondance **<nom –> adresse processeur, nom local>**. Soit une diffusion de la requête permet d'atteindre le serveur ; la requête est ignorée par les sites non concernés. Un mécanisme de cache est nécessaire pour éviter la sur-

charge du réseau due à la diffusion. Il stocke les adresses des serveurs les plus fréquemment utilisés. Lorsqu'un accès est fait à un serveur, sa localisation est mise dans ce cache. Ainsi lors des accès suivants, son adresse sera trouvée dans le cache, il ne sera pas nécessaire de faire la diffusion. Une autre solution pour résoudre les noms est d'utiliser un serveur de noms connu de tous les clients. Il fournit les localisations des autres serveurs à la demande. Là aussi, un cache permet de limiter le nombre des requêtes au serveur de nom. On peut imaginer un compromis de ces deux solutions de base. Un problème annexe est celui de l'évolution dynamique de l'ensemble des serveurs qui nécessite un maintien de cohérence des correspondances nom-localisation. Nous venons que ce problème peut être contourné dans le cadre de notre noyau.

La couche acheminement de messages

Cette couche se charge de faire le transfert des messages, c'est à dire, **expédier**, **recevoir** et **réexpédier**. C'est à ce niveau que le multiplexage des moyens de communication et le **routage** sont traités.

Les situations suivantes peuvent se présenter lors de l'émission ou la réception d'un message sur un processeur :

- **émission** : une fois le message en possession du **noyau de communication**, deux cas se présentent : a) le **processeur destinataire est celui originaire du message** : dans cette situation, le message sera copié localement. b) le **processeur destinataire** est **distant** : le message doit être acheminé par le réseau de communication. Dans ce cas, il faut trouver la route pour atteindre le processeur spécifié et émettre le message sur cette route.

- **réception** : à la réception d'un message deux cas se présentent :
  a) **le message est destiné au processeur récepteur** : le message a donc atteint le processeur destinataire. Il est transmis au processus destinataire. b) **le message n'est pas destiné au processeur récepteur** : il doit être reémis vers un autre processeur. Pour cela, il faut trouver la route vers le processeur destinataire, puis émettre le message à nouveau.

La fonction de calcul des routes **(routage)**[Gon91], [Ray91] est un point important de la couche acheminement. Elle consiste à calculer les chemins de sortie d'un noeud en fonction du noeud destinataire. Cette fonction peut être calculée ou enregistrée dans une **table de routage**. La fonction de routage peut être une constante architecturale du système matériel ou peut être différente selon les versions de système (configuration statique) ou les programmes exécutés (configuration dynamique). Une fonction de routage est caractérisée par des propriétés diverses, parmi lesquelles :

- **déterministe** : on emprunte toujours la même route entre deux processeurs,
- **adaptatif** : on détermine la route en fonction de critères de congestion,
- **sans interblocage/sans famine.**

Indépendamment de la fonction de routage, il y a plusieurs façons d'acheminer un message. Une technique courante est le "store and forward" (stocker et réexpédier). Elle consiste en acheminer un message d'un processeur à autre en stockant le message avant sa reémission. Cette technique a l'inconvénient de faire de la mémoire une ressource importante. Enfin, même dans le cas où le lien de sortie est disponible, la reémission du message est seulement faite après la réception complète du message.

La technique "wormhole routing"[Dal87] consiste à diriger le message directement du lien d'entrée au lien de sortie, dès que l'on acquis l'adresse du site récepteur. Ainsi, un message traverse un processeur sans être stocké complètement. Le problème de cette technique apparaît quand le réseau est surchargé : si le lien de sortie n'est pas libre, il y a une attente de sa libération. Pendant ce temps d'attente, le lien d'entrée n'est pas libéré et il en est de même pour tous les liens/sites en cours de traversée par le message (blocage d'un circuit).

Notre choix pour la réalisation de notre environnement parallèle s'est porté sur une architecture à Micro Noyau. Dans ce qui suit nous allons décrire le Micro Noyau Parallèle puis nous décrirons les serveurs réalisant notre **environnement d'exécution parallèle**.

4.2 Le Micro Noyau Parallèle

Le Micro Noyau Parallèle est une évolution d'un noyau plus simple qui a été développé dans la cadre du projet PLoSys comme support d'une implantation parallèle du langage Prolog[Fav92]. Cette version initiale a été réorganisée et étendue pour servir de base à un **environnement de programmation parallèle autonome**. Le Micro Noyau Parallèle est constitué par un ensemble de Micro Noyaux locaux communicants, un sur chaque noeud (Transputer) de la machine, et les services de chaque noeud sont accessibles par des appels de procédure à distance (RPC[Bir84]).

4.2.1 Le Micro Noyau local

Le Micro Noyau local possède des opérateurs de base qui permettent la **création de processus**, la **synchronisation entre processus** et la **gestion de la mémoire**. Un minimum de service **d'entrée/sortie** est offert. Cet ensemble de fonctions sont incluses dans le langage C//, extension concurrente du langage ANSI C. Ce compilateur a été développé spécifiquement pour les besoins du projet PLoSys où il a servi à la réalisation d'un système Prolog parallèle. Il offre des constructeurs à la CSP (communication de message par canaux typé, alternatives gardées) et des opérateurs de contrôle des entités canaux et processus.

La **création de processus** est exécutée localement et les processus créés ont une relation de filiation avec le créateur. Cette relation de filiation peut être **synchrone** ou **asynchrone**. La relation **synchrone** utilise un schéma du type **cobegin/coend**. La relation **asynchrone** est obtenue en utilisant directement les primitives de création et démarrage de processus sans le parenthèsage (**cobegin/coend**). La création d'un processus se fait avec

```
processus-id p ;
p = p_new(stack_size ,  process_code , args, size)
```

Le démarrage est fait par la procédure **p_run (p)** où p identifie un processus créé par une opération **p_new**. La relation de filiation **synchrone** est indiquée par les constructions **p_begin_par** et **p_end_par**. La construction **p_begin_par** indique le début d'une séquence d'opérations **p_new**. La construction **p_end_par** indique la fin d'une séquence de création de processus. Le processus qui l'exécute est bloqué, et tous les processus créés par des **p_new** à partir du dernier **p_begin_par** sont démarrés. On montre par la suite un exemple de programme écrit dans le langage C//.

```
#include <process.h>
```

```
#include <stdio.h>
#include <chan.h>

chan int c ;
p0 (int n)
{
  int i ;
  for(i=0;i<n;i++)
     out(c, i) ;
}
p1 (int n)
{
  int i, k ;
  for(i=0;i<n;i++)
     k = in(c) ;
}
main()
{
  p_begin_par() ;
    p_new(10000 , "producteur", p0 , sizeof(int), 10) ;
    p_new(10000 , "consommateur, p1 ,sizeof(int), 10) ;
  p_end_par() ;
}
```

Cette exemple est composé par trois processus : **main**, **p0** et **p1**. Le processus **main** crée **p0** et **p1** en utilisant la relation synchrone. Le premier paramètre de l'opération **p_new** indique la taille de la pile, le deuxième est l'identificateur par lequel le processus est connu de façon interne, et le troisième est l'identificateur de la procédure qui sera démarrée. Ensuite, ily a la taille de l'argument que le processus recevra, et finalement l'argument (dans l'exemple 10). Les processus **p0** et **p1** communiquent par le canal **c**. Le processus **p0** exécute 10 fois l'opération **out** d'une valeur entier dans le canal ; le processus **p1** récupère la valeur du canal par l'opération **in**.

Le langage C// support un constructeur, **alt**, qui correspond à la notion d'alternative définie dans CSP. Une instruction **alt** est constituée par un ensemble de **gardes** et **d'alternatives**. Les **gardes** indiquent les conditions qui doivent être satisfaites pour exécuter l'action associée. Une action d'une **alternative** est un ensemble d'instructions C, lié à une garde. L'exemple suivant illustre l'utilisation de l'instruction **alt** avec des gardes. Dans cette exemple, nous avons une garde, composée par une expression logique et un canal, pour chacun des canaux physiques du Transputer, et une garde sur une horloge. La garde sur l'horloge est sélectionnée si le délai d'attente est dépassé. Pour qu'une autre garde soit sélectionnée,

elle doit satisfaire l'expression logique et le rendez-vous sur le canal associée.

```
alt {
    guard exp1 , canal1 :
          i = in (canal1) ;
          ...
          break ;
    guard exp2 , canal2 :
          j = in (canal2) ;
          ...
          break ;
    guard exp3 , canal3 :
          k = in (canal3) ;
          ...
          break ;
    guard exp4 , canal4 :
          k = in (canal4) ;
          ...
          break ;
    guard delay(timer,30) :
          /*delay d'attente depasse*/
          ...
          break ;
    }
```

Le langage offre aussi des primitives pour la gestion des priorités des processus. La **synchronisation** entre processus est assurée par des sémaphores. Les opérations **malloc** et **free** servent pour l'allocation et la libération de la mémoire.

Le noyau local est lui même programmé en C//. Il est organisé selon le modèle client/serveur. La **gestion de processus**, la **gestion de la mémoire** et les **entrées/sorties** sont réalisées par des serveurs noyaux. Par la suite, nous dirons **processus noyau** ou **serveur noyau** pour parler d'un processus ayant en charge certaines fonctions de notre noyau parallèle.

4.2.2 Le Micro Noyau Parallèle

L'environnement d'exécution a été réalisé par un ensemble de Micro Noyaux Coopérants, un pour chaque processeur. Selon la philosophie client/serveur, les services de ces noyaux sont accessibles par un mécanisme **RPC synchrone**. Un serveur

pourra être réalisé, sans restrictions, par un processus maître créant autant d'esclaves que nécessaire. Cette solution présente deux caractéristique importants : **concurrence entre les opérations d'un serveur** et **élimination de l'interblocage**.

L'adressage des serveurs

Le problème de l'adressage dans notre contexte est plus simple que dans le cadre d'un système distribué. En effet, les fonctions des serveurs que nous avons à écrire sont de deux types :

- celles créant les objets de notre environnement sur un site donné (ex. porte, barrière, ...),
- celles opérant sur un objet donné.

Par conséquence, on peut voir notre environnement comme composé de serveurs répliqués sur chaque site et responsables des opérations sur les entités de leur site. Nous pouvons donc choisir une solution simple où un serveur reçoit une identification locale identique sur tous les sites. Les identifications locales et globales sont alors identiques. Tout objet créé reçoit une identification qui comporte une partie contenant l'adresse du site support de l'objet. Il est donc à la charge de la procédure stub client de déterminer le site du serveur à utiliser. Pour les procédures de création, ce site est un argument direct de la procédure. Les autres opérations ont un argument identifiant un objet et cet identificateur encode l'adresse du site support de l'objet. Le site étant toujours donné directement ou non par un argument, le serveur est déterminé par la fonction appelée. Cette solution est satisfaisante pour un domaine où l'ensemble de serveurs est connu et statique et où on peut déduire la localisation du serveur des paramètres de la procédure.

Les paramètres

Comme tous les langages C, C// supporte le passage de paramètres par **valeur** et par **référence**. Dans une passage par valeur, une copie du paramètre est faite pour la procédure appelée. Les modifications faites sur la copie n'altèrent pas la valeur originale. Par contre, si un paramètre est envoyé par référence, c'est l'adresse de la variable que la procédure appelée reçoit ; les modifications altèrent la variable

originale. Une troisième forme, qui n'existe pas dans C, est la passage par **valeur/résultat**, qui consiste à copier les paramètres pour la procédure appelée et à la fin, copier les résultats pour la procédure appelante.

Dans les machines UMA/NUMA, il est possible de passer un paramètre par "référence" ; ce n'est pas le cas pour une machine NORMA. Les procédures **stub client** et **stub serveur** doivent alors assurer la transparence et permettre aux utilisateurs de faire les appels de manière usuelle. Un client fait un appel à une procédure de façon traditionnelle et envoie les paramètres. La procédure **stub client** crée un message et y copie simplement les paramètres qui sont passés par valeur. Le problème est la passage par **référence**. Une valeur de type adresse n'a pas le même sens dans deux contextes de processeurs différents. Par exemple, l'adresse correspondant à une liste, sur le site client, peut correspondre à un morceau de code sur le site serveur. Une solution est introduire des opérations de lecture/écriture à distance qui permettrait à la procédure appelée d'acquérir les valeurs des paramètres. Une autre est d'interpréter le passage par référence comme un passage par **valeur/résultat**. La procédure stub client fait la copie de la valeur de la variable dans le message et l'envoie au serveur. Ce message est stocké sur le site du serveur, qui appelle la procédure correspondant au service, en donnant l'adresse locale de la variable. Les modifications seront faites sur la copie locale. Le message de réponse qui sera envoyé au client contiendra la valeur modifiée. La procédure stub client, reçoit cette valeur, en fait la copie et assigne l'adresse de cette copie à la variable paramètre.

Dans le cadre de notre environnement, les procédures stub sont relativement simples et ont généralement des paramètres par valeur. Pour les passage par référence, on les transforme en un passage par **valeur/résultat**. Dans ce cas, il est nécessaire de copier complètement une structure de donnée pointée par un paramètre (emballage) et au retour de la reconstruire (déballage).

La définition d'un service chez le client

Un ensemble de constantes numériques définissent les identificateurs locaux-globaux des serveurs :

==========

```
*  Identification des serveurs  *
================================

#define   G_TACHES           0
#define   G_FICHIERS         1
#define   G_COMMUNICATION    2
#define   G_SYNCHRONISATION  3
#define   G_MEMOIRE          4
```

De même que pour chaque serveur, un ensemble de constantes numériques définissent aussi les procédures. Ces constantes sont incluses dans un fichier qui constitue la table de définition des serveurs. Cette table de constantes est utilisée par les procédures stub clients. Le processus client ne connaît les serveurs que par la bibliothèque des procédures stub, c'est à dire, la bibliothèque des opérateurs de notre environnement parallèle. Une partie des opérations du serveur **gestionnaire de synchronisation** est définie par :

```
================================================
*  Identification des operations              *
*  du serveur gestionnaire de synchronisation *
================================================

#define   CREATE         0
#define   DESTROY        1
#define   BARRIER        2
```

Un exemple de procédure stub client est le suivant :

```
=========================
*     stub client       *
=========================
#
# inclusion des constants definissant  les
# operations du serveur
#
BARRIER_CREATE (int processeur, t_barriere * b, int nproc)
 {
   message * m ;
   alloc-message(m) ;  /* m est le buffer du message d'appel */
                       /* et celui du message de reponse */
   /* emballer les parametres */
   m->type = APPEL ;
   m->destinataire.proc = processeur ;
```

```
    m->destinataire.serveur = G_SYNCHRONISATION ;
    m->destinataire.fonction = BARRIER ;
    m->source.proc = my_proc ;
    m->source.processus = my_processus_id ;
    m->source.m-reponse = m ;
    m->donnees.param[0] = nproc;
    send_rec(m) ;
    /* deballer les parametres */
    tid = m->donnees.param[0] ; /* au retour param[0] contient b */
    free-message(m) ;
    return() ;
}
```

Cette procédure fabrique le message de demande de création d'une barrière. Elle alloue une structure de donnée de type message (m) pour contenir l'appel, et que aussi contiendra la réponse. Elle emballe les paramètres dans les champs spécifiques. Le premier identifie le type du message, APPEL dans le cas de l'exemple. Ensuite, elle remplit les champs identifiant le destinataire (processeur, serveur et fonction). Le champ m->source.proc identifie le processeur originaire du message, m->source.processus identifie le processus émetteur, et m->source.m-reponse contient l'adresse du message m, qui sert aussi pour la stockage des résultats. Elle met aussi dans le message les arguments de la fonction qui sera exécutée par le serveur : nproc identifiera, dans l'exemple, le nombre de processus qui participeront de la barrière. Le message est alors transmis au serveur (send_rec). A la réception de la réponse, les résultats sont déballés et le tampon du réponse desalloué.

La définition d'un service chez le serveur

Un serveur est décrit par un numéro valide sur tous les sites puisque les serveurs sont répliqués. Le noyau de communication sur chaque site possède une table de serveurs. Chaque entrée dans cette table identifie un serveur et contient la file des requêtes enregistrées et non acquises. L'enregistrement du serveur dans la table est fait de la façon suivante : le serveur s'identifie comme serveur, et reçoit l'adresse de son entrée dans la table de serveurs. Toutes les fois qu'un message arrive, le noyau de communication analyse son en-tête pour déterminer le serveur destinataire, et le met dans la file du serveur concerné. La file préserve l'ordre de réception des messages ; leur acquisition sera effectuée selon cet ordre (FIFO) par le serveur.

Lorsqu'un serveur est prêt pour accepter des requêtes, il exécute la primitive **dequeue (my_entry, m)** qui le bloque s'il n'existe pas de messages dans sa file. Si par contre la file n'est pas vide, il prend le premier message et crée un processus "esclave" pour l'exécuter et se remet en attente de nouvelles requêtes. Le processus esclave exécute alors l'opération demandée et renvoie les résultats au client. Les arguments nécessaires pour l'exécution de l'opération sont extraits (déballés) du message par le processus "esclave" qui emballera les résultats à la fin d'exécution de la requête. Nous présentons, à la suite, l'organisation générale du serveur **gestionnaire de tâches** et du processus "esclave" créé pour l'exécution d'une requête.

```
====================
*   serveur maitre  *
====================

main()
{
  message * m ;
  /* my_entry contiendra l'adresse de la file
      du serveur de gestion de taches G_TACHES */
  my_entry = new_server(G_TACHES) ;
  for (;;) {
      /* extraction du premier message de la file */
      m = dequeue (my_entry) ;
      /* creation d'un esclave pour le traitement de de la requete */
      p_run(p_new(esclave, m)) ;
      }
}
====================
*     esclave       *
====================

esclave (message * m)
{
   switch (m->fonction){
          case EXECTASK :
                  "deballer les parametres"
                  "appeler la procedure qui cree la tache et
                  emballe son identificateur dans m"
                  reply (m) ;
                  break;
```

```
        case EXECTHREAD :
                "deballer les parametres"
                "appeler la procedure qui cree le processus et
                emballe son identificateur dans m"
                reply (m) ;
                break;
        ...
        }
}
```

Le message

Le message est l'entité de communication entre les clients et les serveurs. C'est un ensemble de données qui définissent une requête ou une réponse. Un appel d'une fonction système est transformé en un message qui sera envoyé au serveur spécifique qui devra exécuter le service et renvoyer les résultats au client. Le message doit donc contenir toutes les informations nécessaires à la réalisation de la requête et à la réception des résultats. Il doit spécifier la **localisation** et l'**identification** du **serveur destinataire**, le **service demandé**, et les **paramètres** nécessaires à l'exécution du service. De manière à permettre au serveur d'envoyer la réponse avec les résultats, le message doit contenir la **localisation** et l'**identification** du client demandeur du service, de même que le **tampon** pour le stockage des résultats.

Dans notre système, le message possède trois parties : un **type**, une **en-tête**, de taille fixe, et les **données** (de taille variable). Comme nous avons un seul protocole de communication (RPC), les messages peuvent être de deux types : APPEL et REPONSE. L'en-tête dépend du type et comprend deux champs : **source** et **destinataire**. La figure suivante montre le format d'un message d'appel. Dans le champ

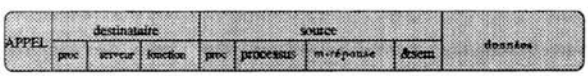

Figure 4.8: Format d'un message d'APPEL.

destinataire, **proc** identifie le processeur destinataire, **serveur** désigne le processus destinataire, qui est toujours un **serveur** et **fonction** spécifie la procédure appelée. Le champ **source** du message identifie l'émetteur : **proc** identifie le processeur et **processus** identifie le processus client. Chaque message contient deux adresses :

l'adresse du **sémaphore** sur lequel le processus client est bloqué en attendant la réponse (cf. send_rec qui on présentera par la suite), et l'adresse du buffer destiné à la réponse (**m-réponse**). Dans le cas d'une REPONSE, le destinataire est le processeur client. Le processus client est retrouvé grâce aux adresses de tampon de réponse et de sémaphore qui sont renvoyés sans modification par le serveur. Le format d'un message **REPONSE** est décrit dans la figure 4.9. La partie **données** des

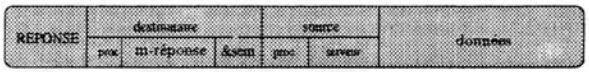

Figure 4.9: Format d'un message de REPONSE.

messages contient les arguments d'un APPEL, ou les résultats d'une REPONSE. Un premier champ de cette partie défini le nombre d'arguments (ou résultats) et chaque argument (ou résultat) est représenté par sa **taille** et son **adresse**.

4.2.3 Le micro noyau de communication

Un point important du Micro Noyau Parallèle est la fonction de communication. Elle constitue le **Micro Noyau de Communication**. Il comprend deux couches : **protocole** et **acheminement de messages**. La couche **protocole**, implante un protocole client/serveur de base. Tous les protocoles de communication de notre environnement de programmation parallèle seront implantés alors par des serveurs spécifiques. Ce protocole minimum est une variante du protocole RPC. Nous pensons que cette solution est intéressante car elle simplifie la couche **protocole**. Le noyau de communication en devient plus petit et, par conséquent, plus facile à réaliser.

L'organisation du noyau de communication

Dans les systèmes traditionnels, le niveau utilisateur demande un service au noyau qui l'exécute et renvoie les résultats, par exemple, en utilisant les registres de la machine. Dans l'organisation d'un Micro Noyau, le noyau ne transmet que la demande et la réponse. Le service est réalisé par un serveur spécifique. Nous allons suivre cette philosophie en exploitant au mieux les possibilités de parallélisme matériel dues aux processeurs de communication du Transputer dans l'organisation du noyau de communication. Une possibilité d'organisation est d'utiliser un

seul processus noyau "**émetteur**" dans le noyau de communication. Pour envoyer un message, tous les processus clients ou serveurs le font par l'intermédiaire de ce processus noyau qui gère une file de messages. Il prend le premier message de la file, calcule la route pour atteindre le destinataire et émet ce message sur le lien de sortie correspondant (figure 4.10). Ce processus noyau émetteur unique est un

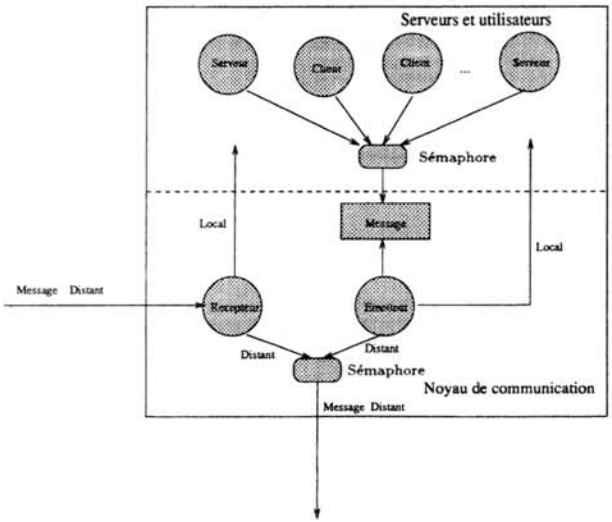

Figure 4.10: Noyau de communication avec un seul émetteur.

goulot d'étranglement qui interdit l'accès parallèle aux canaux physiques. Une meilleure solution est chaque processus client ou serveur émettre lui même son message. L'avantage de cette stratégie est de paralléliser au maximum les envois de messages issus des différents processus. Le seul retard possible apparaît dans le cas d'un envoi distant, où le lien physique de sortie est déjà occupé par un autre processus émetteur. De plus, les messages locaux sont envoyés sans délai au destinataire. La figure 4.11 illustre cette organisation. Si l'on recherche un parallélisme maximum, la réception d'un message sera aussi organisé autour d'un processus noyau **récepteur** par lien, qui soit transmettra le message au processus client ou serveur concerné soit reémettra ce message.

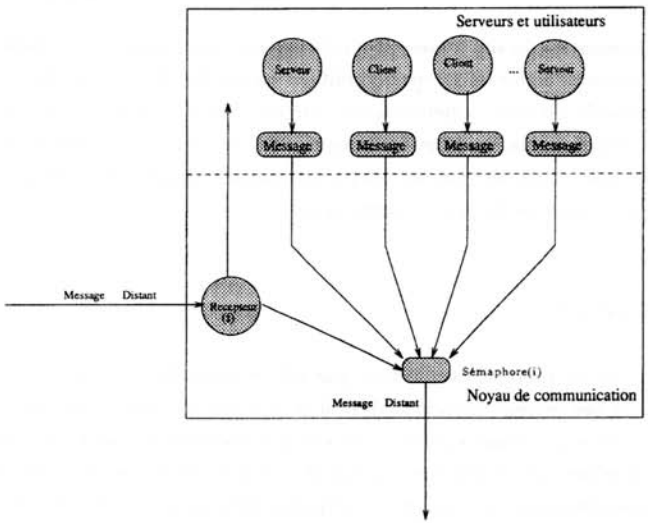

Figure 4.11: Noyau de communication avec plusieurs émetteurs.

1 La couche protocole

Comme nous avons déjà dit en 4.2.3, le seul protocole implanté pour la couche protocole est un protocole de type **client/serveur** variante du protocole RPC. Les autres protocoles sont implantés par des serveurs spécifiques. Un serveur reçoit les messages de clients, soit locaux, soit distants. Sur le site serveur, ces messages sont stockés dans la file FIFO du serveur. Pour exécuter une requête, le serveur appelle une procédure qui lui retourne le premier message de cette file, ou le bloque s'il n'y a pas de messages (dequeue).

L'architecture de la couche protocole

La couche protocole doit permettre au client d'envoyer la requête au serveur et attendre pour les résultats, et au serveur de recevoir les requêtes et envoyer les résultats. Du côte client, pour envoyer le message le client émet directement. A la réception, un message est mis dans la file FIFO du serveur par un proces-

sus **récepteur** du site serveur. Du côte serveur, une procédure récupère le premier message de la file, le serveur le traite et envoie les résultats au client. La couche protocole est donc organisée autour de ces trois fonctions : **l'envoi de la requête**, par un client, la **réception** de la requête du côte serveur, et **l'envoi de la réponse**. Par ailleurs, les serveurs sont des processus normaux qui en appelant une procédure spéciale se déclarent comme serveur.

La réalisation

La couche protocole est définie par quatre procédures : **send-rec**, **dequeue**, **reply** et **new-serveur**. La primitive **send-rec** est exécutée par les processus clients, et les autres par les **serveurs**. Deux procédures intermédiaires sont utilisées. **alloc_link** et **free_link** permettent l'acquisition d'un lien en exclusion mutuelle. **out_link** est l'opérateur d'émission sur le lien physique. **get_route** est la fonction de routage et sera décrite plus loin.

1. **send-rec** : la primitive **send-rec** est utilisée pour **envoyer le message** généré par une procédures **stub client** au serveur concerné, et **attendre la réponse**. La structure générale de cette procédure est la suivante :

```
send_rec(message * m)
{
  semaphore s ;
  channel c ;
  /* preparation de la synchronisation d'attente de la reponse */
  sem_init(s, 0) ;
  m->source.sem = &s ;
  /* emission*/
  lp = get_route(m->destinataire.proc) ;
  alloc_link(lp) ;
  out_link(lp, m) ;
  free_link(lp) ;
  /* attente de la reponse */
  sem_P(m->source.sem) ;
}
remarque : on n'a pas ici decrit le cas d'une communication locale
           (cas ou le processeur destinataire est le processeur local).
```

Cette procédure reçoit l'adresse du message comme paramètre, obtient le lien de sortie pour atteindre le destinataire, par un appel à la procédure **get_route**, alloue le lien physique (**alloc_link**), émet le message (**out_link(lp, m)**), libère le lien (**free_link**), et se bloque sur le sémaphore du message en attente de la réponse.

2. **Dequeue** : cette procédure est exécutée par le serveur pour recevoir une requête. Toutes les requêtes envoyées à un serveur sont mises dans la file du serveur, et cette procédure extrait la première requête de la file. Un appel à cette procédure est bloquant s'il n'existe pas de messages dans la file. Elle a comme paramètre l'adresse du descripteur du serveur, qui contient la file du messages, et retourne l'adresse du premier message de cette file. Elle utilise deux sémaphores, un qui la bloque si la file est vide, et un autre pour l'exclusion mutuelle dans la manipulation de la file.

```
message * dequeue(descripteur * my-desc)
{
  message  * m ;
  sem_P(&(my-desc->consom)) ; -- attendre l'arrivee d'un message --
  sem_P (&(my-desc->mutex)) ;
  "recuperer le premier message de la file du serveur"
  sem_V (&(my-desc->mutex)) ;
  return(m) ;
}
```

3. **reply** : la procédure **reply** est utilisée par le serveur pour envoyer la réponse au client. Alors que la procédure **send-rec** qui est bloquante, la procédure **reply** ne bloque pas le serveur : la réponse est envoyée et le serveur reprend l'exécution. Elle reçoit, comme paramètre, le message original du client, mais dont la partie de **données** contient les résultats de la requête. Ils ont été mis par la procédure qui a exécuté le service. Le code de la procédure **reply** est le suivant :

```
reply(message * m)
{
  /*preparer message*/
  m->type = REPONSE ;
  m->destinataire.proc = m->source.proc ;
  m->destinataire.m-reponse = m->source.m-reponse ;
  m->destinataire.sem = m->source.sem ;
```

```
  m->source.proc = m->destinataire.proc ;
  m->source.serveur = m->destinataire.serveur ;
  /* emission*/
  lp = get_route(m->destinataire.proc) ;
  alloc_link(lp) ;
  out_link(lp, m) ;
  free_link(lp) ;
}
```

Cette procédure débute par la permutation de l'en-tête car les rôles source et destinataire sont inversés. Le type du message est changé. Ensuite, la route et le lien correspondant sont acquis et l'émission de la réponse à lieu.

4. **new-serveur** : permet l'installation d'un serveur dans le Micro Noyau Parallèle. Un serveur dans notre système est lancé comme un processus normal, et pour être connu du système il doit s'identifier comme serveur et obtenir l'entrée de la table de serveurs qui le décrit. Cette procédure initialise la structure de données qui décrit le serveur. Cette structure est composée par :

   - un sémaphore qui indique l'existence de messages dans la file,
   - un sémaphore pour l'exclusion mutuelle lors de la manipulation de la file,
   - l'adresse du premier message de la file, et
   - l'adresse du dernier message de la file.

   Elle reçoit comme paramètre l'identification du serveur (un entier), initialise les sémaphores utilisés dans la gestion de la file du serveur et les pointeurs de premier et dernier message (comme vides). Puis, elle retourne l'adresse de l'entrée au serveur. Le code de la procédure est le suivant :

```
serveur-descripteur * new_server(int server_id)
{
  serveur-descripteur sd ;
  if (server_id < 0 || server_id >= NB_SERVERS)
```

```
        return(NULL) ;
    sd = &(services[ server_id ]) ;
    sem_init (&(sd->mutex) , 1) ;
    sem_init (&(sd->consom) , 0) ;
    sd->head = NULL ;
    sd->tail = NULL ;
    return(sd) ;
}
```

2 La couche acheminement de messages

La couche **d'acheminement de messages** est composée de quatre processus **récepteurs** (un pour chaque lien physique du Transputer) et par une **fonction de routage**.

La réception de messages

Lorsqu'un message adressé au processeur arrive, il doit être transmis à un processus client (REPONSE) ou à un serveur (APPEL). Dans le cas d'une REPONSE, le champ **m-réponse** de l'en-tête du message contient l'adresse du tampon, chez le client, qui doit stocker les résultats. S'il en a, le processus **récepteur** les lit sur le lien physique et les stocke dans ce tampon. Ensuite, il réveille le client, par une opération V sur le sémaphore, dont l'adresse fait partie de l'en-tête du message. Le client reprend l'exécution dans la procédure stub client. Cette procédure déballe les résultats (s'ils existent) du tampon, et termine en libérant le tampon. Si le message qui arrive est un APPEL, le processus **récepteur** identifie le serveur destinataire dans la table des serveurs, il alloue un tampon, lit le message du lien physique et le stocke dans le tampon. Ensuite, il met le tampon dans la file du serveur et le réveille, si nécessaire.

L'émission de messages

Elle est faite par les procédures **send_rec** et **reply**. Le processus émetteur la transmet directement au serveur, sinon sur le réseau de processeurs. Il utilise alors la fonction de routage.

L'acheminement des messages

Le processus **récepteur** doit réexpédier un message qui est arrivé et n'est pas

destiné au processeur. De la même manière, une requête générée par un client pour un serveur distant, ou une réponse à un client distant, doivent aussi être acheminées. Pour **réexpédier** un message, le processus **récepteur** doit émettre sur le lien du sortie à partir duquel le processeur cible est accessible. Pour envoyer une requête ou une réponse, le processus émetteur l'émet sur le lien de sortie par lequel le destinataire est accessible. Dans le deux cas, il faut consulter la **table de routage**. Les accès à la table de routage et comme à un lien de sortie doivent être faits en exclusion mutuelle.

Pour l'**acheminement de messages**, nous avions deux possibilités : la première est de considérer que le routage est intrinsèque à la machine cible et donc que notre noyau doit utiliser la fonction de routage de la machine. C'est une solution réaliste pour une machine parallèle à maillage statique. C'est n'est pas le cas du Supernode qui peut être reconfiguré dynamiquement. Nous avons donc choisi de prendre compte de cette possibilité en fixant simplement un mode de routage ("wormhole routing"), mais en laissant une fonction de routage reconfigurable. Cette fonction de routage se présente comme une simple table indicée par le numéro du processeur cible et qui fournit le numéro de lien de sortie. La construction d'une table "bien formée" n'est pas de la responsabilité du Micro Noyau. C'est celle du concepteur de l'application parallèle.

Lorsque le Micro Noyau est chargé sur un processeur, il initialise la table de routage. Chaque processeur est identifié par un numéro logique que le Micro Noyau local reçoit comme paramètre quand il démarre. Le numéro lui permet de sélectionner la table de routage initiale. Le routage peut évoluer au gré de l'application, soit pour des raisons liées au contrôle du flux des communications, soit parce que la machine est reconfigurée physiquement. Une configuration locale de la table de routage est purement logique et n'a aucune incidence sur la configuration matérielle. Il est à la charge de l'application de s'assurer de la cohérence des tables locales au cours de leurs évolutions. Le seul mécanisme offert permet la modification atomique d'une table locale. Par ailleurs, il est aussi à la charge de l'application de coordonner le routage et la configuration physique de la machine.

Dans notre implantation, la stratégie de routage utilisée est une variante du "wormhole routing". Le message n'est pas complètement stocké sur un processeur. L'en-tête est envoyé d'un processeur vers l'autre, et le message est lu par blocs. Avec cette méthode, à un instant donné le message peut être réparti sur les processeurs du chemin allant de la source à la cible.

L'implantation

L'implantation de la couche **acheminement** se réduit à la réalisation de la **fonction de routage**, et des processus **récepteurs**. Deux procédures manipulent la file du serveur : une est exécutée par le serveur et récupère le premier message de la file (**dequeue**, présentée auparavant), et l'autre (**enqueue**) est exécutée par les processus récepteurs et émetteurs et ajoute un message dans la file. La procédure **enqueue** reçoit deux paramètres : l'adresse du descripteur du serveur et l'adresse du message à insérer dans la file. Après avoir ajouté le message (en exclusion mutuelle), elle signale au serveur l'existence de messages dans la file (opération V sur le sémaphore où le serveur est bloqué en attente de messages) :

```
enqueue(serv-descripteur * sd, message * m)
{
  sem_P(&sd->mutex) ; /* exclusion mutuelle de la file du serveur */
  "inserer le message m  dans la file"
  sem_V(&sd->mutex) ; /* liberer la file du serveur */
  sem_V(&sd->consom) ; /* signaler l'arrivee d'un message au serveur */
}
```

La gestion de routage comprend trois procédures : **get-route**, qui reçoit comme paramètre un numéro de processeur et retourne le lien de sortie pour lui accéder ; **alloc-link (lien)** qui alloue le lien physique pour une utilisation exclusive, et **free-link (lien)** qui libère le lien physique. La procédure **get-route** est décrite ci-dessous. Les deux autres se résument, respectivement, à une opération P et à une opération V sur le sémaphore associé au lien.

```
int  get-route(processeur p)
{
  lien ls ;
  sem_P(&sem-route-table) ;
  ls = route-table[p] ;
  sem_V(&sem-route-table) ;
  return (ls) ;
}
```

Nous allons préciser maintenant des notations que nous utiliserons pour présenter le squelette du processus **récepteur**.

in-msg (lien-entrée, en-tête) : indique la lecture de l'en-tête d'un message du lien physique **lien-entrée**. Le processus est bloqué jusqu'à ce qu'un message arrive.

recevoir-appel (lien-entrée, en-tête) : cette fonction alloue une structure du type message, stocke l'en-tête déjà lu dans le message alloué, lit les argument sur **lien-entrée** et les stocke dans le message. Puis, elle décode l'identification du serveur

et appelle la procédure **enqueue** avec l'identification du serveur et l'adresse du message alloué, qui sera placé dans la file du serveur.

recevoir-réponse (lien-entrée, en-tête) : cette fonction lit les arguments d'un message du **lien-entrée** et les stocke dans le tampon dont l'adresse est le champ **en-tête->destinataire.m-réponse**. Ensuite, elle réveille le client, par une opération V sur le sémaphore dont l'adresse est dans l'entête (**en-tête->destinataire.sem**).

acheminer (lien-entrée, lien-sortie, en-tête) : émet sur **lien-sortie** l'en-tête reçu, et lit les arguments d'un message sur **lien-entrée** et les émet sur **lien-sortie**.
Le modèle de processus **récepteur** est le suivant :

```
============================
*  Processus recepteur     *
============================

recepteur (int mon-processeur-numero, lien lien-entree)
{
  t-entete entete ;
  lien lien-sortie ;
  processeur p ;
  for (;;) {
      /* lire l'en-tete du message du lien-entree */
      in-msg (lien-entree, en-tete)  ;
      proc = en-tete->destinataire.proc ;
      if (proc == mon-processeur-numero) {
         if (en-tete->type == APPEL)
            recevoir-appel(lien-entree, en-tete) ;
         else
            recevoir-reponse(lien-entree, en-tete) ;
      else {
            lien-sortie = get-route (proc) ;
            alloc-link (lien-sortie) ;
            acheminer (lien-entree, lien-sortie, en-tete) ;/* wormhole */
            free-link (lien-sortie) ;
            }
      }
}
```

4.2.4 Bilan

Ainsi, notre Micro Noyau se présente comme une structure simple, modulaire et régulière. Il es adaptable à toute configuration. Nous allons montrer qu'il est suffisant pour construire l'environnement d'exécution défini dans le chapitre 3.

4.3 Les services

Chacune des fonctions de **communication**, de **synchronisation** et de **gestion de processus** de notre environnement parallèle sont implantées par un module à part, le **serveur** qui est répliqué sur chaque processeur. Toutes les interactions entre les clients et les serveurs, et entre serveurs, se font par le mécanisme RPC du Micro Noyau. Les primitives (opérations) apparaissent aux programmeurs à travers une bibliothèque de procédures stub. Par leurs intermédiaire, un client fait un appel à un serveur et se bloque, en attendant la réponse. Le serveur n'envoie la réponse qu'au moment où celle-ci est calculée, débloquant aussi le client. De façon identique, lorsqu'un serveur a besoin de la coopération d'un autre serveur pour la réalisation d'un service, il envoie une requête, et attend la réponse.

Trois serveurs réalisent les différents services de notre environnement :

Service de communications : dans notre système, les processus communiquent de manière **indirecte**, en utilisant les **portes**. La communication peut être **synchrone** ou **asynchrone**, et il existent deux types de portes : **Data** et **Message**. Le **service de communication** est responsable de la gestion des **portes**, et des primitives associées.

Service de synchronisation : les processus composants d'une application parallèle peuvent synchroniser leurs actions par le mécanisme de **barrière**. Le **gestionnaire de synchronisation** gère le mécanisme **barrières**, et les opérations associées.

Service de tâches et processus : il exécute les requêtes liées à la gestion de processus, qu'il reçoit de clients locaux ou de serveurs distants.

Par la suite nous ne décrirons pas les procédures stubs clients dont un exemple a été donné en 4.2.2. Nous décrirons essentiellement les points spécifiques de chacun

des services sachant que tous sont conformes à un même modèle d'organisation.

4.3.1 Organisation générale des services

Chaque service est implanté par un serveur répliqué sur chaque processeur de la machine parallèle. Les serveurs sont des processus normaux qui se déclarent comme serveur par **new-serveur** (présenté en 4.2.3). Ils attendent alors des messages. Pour recevoir un message, le serveur exécute la fonction **dequeue (my_entry, m)** qui retourne le premier message de sa file ou le bloque s'il n'existe pas de message. Quand il prend un message, il crée un processus "esclave" pour exécuter la requête et retourne à sa boucle d'attente de requêtes. Le processus esclave exécute l'opération et renvoie les résultats au client. Tous les arguments nécessaires pour exécuter l'opération sont contenus dans le message. L'organisation générale d'un serveur et du processus "esclave" créé pour l'exécution d'une requête est la suivante :

```
=====================
*   serveur maitre  *
=====================

main()
{
  message * m ;
  my_entry = new_server("serveur-id") ;
  for (;;) {
      /* extraction du premier message de la file */
      m = dequeue (my_entry) ;
      /* creation d'un processus esclave
      pour le traitement de de la requete */
      p_run(p_new(esclave, m)) ;
      }
}
=====================
*     esclave       *
=====================

esclave (message * m)
{
   switch (m->destinataire.fonction){
     case OPi : "deballer les parametres"
                do_opi(args, result) ; /* execute le service */
                                       /* et emballe les resultats dans m */
```

```
                reply (m) ;
                break;
        case OP2 : "deballer les parametres"
                do_op2(args, result) ; /* execute le service */
                                       /*et emballe les resultats dans m */
                reply (m) ;
                break;
        ...
        case OPn : "deballer les parametres"
                do_opn(args, result) ; /* execute le service */
                                       /*et emballe les resultats dans m */

                reply (m) ;
                break;
        }
}
```

Comme l'on crée un esclave par requête, il suffira de synchroniser les esclaves entre eux pour synchroniser les clients (qui attendent la réponse de leur esclave). Par la suite, nous ne décrirons que les procédures de service **do_op**$_i$ exécutées par les processus esclaves.

4.3.2 Interaction entre serveurs

Pour toutes les primitives ne faisant intervenir qu'une entité (porte ou processus), l'exécution ne met en cause qu'un client et un serveur. Ce n'est plus toujours le cas dès qu'une primitive fait intervenir une liste d'entités. Deux cas de figure sont possibles :

- la liste de diffusion,
- la liste d'attente.

Dans le premier cas, il s'agit d'effectuer la même opération sur plusieurs entités. C'est le cas de la réplication de processus ou de la diffusion d'une valeur sur des portes. Ce cas se ramène à un ensemble d'opérations simples sans interaction. Il n'en est pas de même pour les listes d'attente de processus, les listes d'attente sur des portes messages et les listes d'attente d'acquisition de portes données qui nécessitent une interaction entre les serveurs gérant les différentes entités de la

liste. Ainsi, une attente d'une communication en provenance de plusieurs portes implique-t-elle :

1. la diffusion de l'attente sur une porte à tous les serveurs concernés,
2. si le serveur déduit que la requête peut être satisfaite, il doit le signaler de façon à :
   - choisir une porte parmi toutes les portes prêtes (choix non déterministe),
   - confirmer l'acceptation ou le refus du message proposé par les différents parties (serveurs).

On voit ici la nécessité d'un dialogue complexe entre le client et les différents serveurs concernés. Nous traiterons les différents cas spécifiques selon un principe identique, soit de transférer la requête à un serveur de diffusion, soit de transférer la requête au serveur local responsable du type d'entité constituant la liste. C'est le serveur local qui coordonnera le dialogue avec les serveurs distants de même type.

4.3.3 Gestion des entités

Outre un même schéma d'organisation, les serveurs gèrent de façon identique les objets dont ils sont responsables. Il existe trois entités adressables dans notre environnement parallèle : les **portes**, les **barrières** et les **processus**. L'identification de chaque entité contient le site où elle a été créée de manière à permettre sa localisation. Ainsi, toutes les entités sont désignées de la même façon, par une structure de données qui contient :

- **type** : caractérise le type de l'entité,
- **processeur** : c'est le processeur où l'entité a été créée,
- **numéro Interne** : son numéro unique sur ce processeur, et
- **date** : sa date de création.

Toutes les entités peuvent être créées et détruites dynamiquement. Si on considère qu'un numéro local peut être utilisé plus d'une fois pour désigner une entité, la **date** de création permet distinguer une entité par rapport à une autre de même type, de même numéro local, mais créée à un moment différent. La création comme la destruction des entités suivent toujours le même schéma. Un descripteur d'entité est alloué dynamiquement. Il dépend du type d'entité. Ce descripteur est enregistré dans une table (par type), dans une entrée libre (s'il y en a). L'identificateur unique est alors forgé à partir du type, du processeur, de l'indice de la table et de la date de création. Le descripteur est aussi estampillé pour la date. Lors de tout accès à l'entité, un contrôle de validité de l'identificateur fourni est fait en comparant la date portée par l'identificateur et celle portée par le descripteur. A la destruction, le descripteur et l'entrée de la table sont libérés. Les tables des entités sont toutes manipulées en exclusion mutuelle, ainsi que les descripteurs d'entités, du fait de l'accès concurrent par les processus esclaves. Quatre procédures de manipulation sont nécessaires :

- Allouer une entrée, un descripteur et fabriquer une identification unique.
- Acquérir un descripteur en exclusion mutuelle.
- Libérer un descripteur.
- Libérer une entrée sur une table de descripteurs.

4.3.4 Le service de synchronisation

Les serveurs de **synchronisation** gèrent les **barrières**. Les **barrières** constituent un mécanisme qui permet aux processus de synchroniser leurs actions. Dès la création d'une **barrière**, tous les processus la connaissant peuvent l'utiliser. Comme toutes les entités de notre noyau, une barrière est identifiée par un triplet **<numéro processeur/numéro interne/date>**. Son descripteur contient le quorum (nombre de processus qui doivent se synchroniser sur la barrière), le nombre d'opérations **barrière** déjà exécutées et la liste des processus qui sont bloqués sur la **barrière**, ainsi que le nouveau quorum proposé.

Les procédures de service

L'ensemble des requêtes soumises à un serveur de **synchronisation** est montré dans la figure 4.12. Les opérations ne portent que sur une barrière à la fois. A chaque

MESSAGE	PARAMETRES D'APPEL	REPONSE
BARRIER_CREATE	processeur, &b-id, n	b-id
BARRIER_DESTROY	b-id	0 si OK
BARRIER	b-id, nq	rien

Figure 4.12: Les opérateurs de synchronisation.

opération correspond une procédure stub client qui émet la requête à destination du **serveur de synchronisation** portant la barrière. A chacune des requêtes correspond une simple procédure de service réalisant la requête.

do_barrier_create (processeur, b, n) : cette opération crée, sur **processeur** une barrière. Le paramètre **n** indique le nombre de processus qui participeront à la barrière. Le processus esclave alloue et initialise une structure de données qui décrit une barrière, l'enregistre dans la table de barrières locales au processeur, et renvoie au client l'identificateur de la barrière créée.

do_barrier_destroy (b) : la barrière **b** est détruite. Tous les processus clients en attente sur la barrière doivent être réveillés. Le serveur (processus esclave) cherche la barrière **b** dans la table de barrières, et s'il existe des processus esclaves bloqués, il les réveille et détruit la barrière. Les processus esclaves réveillés répondent aux clients.

do_barrier (b, nq) : le serveur incrémente le compteur de processus qui ont déjà exécuté l'opération sur la barrière **b**. Si l'arrivée du processus est la dernière attendue, tous les processus esclaves dans la liste de la barrière sont débloqués, et débloquent à leur tour leur client. Dans le cas contraire, le processus esclave se suspend. **nq** indique le nombre de processus qui se synchroniseront la prochaine fois. Pour le changer, tous les participants doivent être d'accord. Cette vérification est faite par la comparaison des **nq** des différents appels. S'ils sont égaux, le quorum des processus associé à la barrière **b** passe à **nq**.

4.3.5 Le service de gestion des tâches et processus

La gestion de tâches et processus est réalisé par les serveurs de **gestion de tâches et processus**. Comme tous les autres serveurs, ils exécutent des requêtes émises par les clients. La figure 4.13 présente la liste de requêtes que ces serveurs traitent, ainsi que les paramètres de chacune. Les objets tâches et processus sont

MESSAGE	PARAMETRES D'APPEL	REPONSE
LISTEN	non	non
EXECTASK	processeur, "name", &tid, n, arg1,..argn	tid
EXECTHREAD	tid, "proc-nom", n, arg1, ..., argn	tid
ALLWAIT	list-processus	non
ANYWAIT	list-processus	non
TERMINATE	list-processus	non
STATUS	processus-id	0: Vivant 1: Mort 2: Inexistant
EXIT	non	non

Figure 4.13: Les opérations de gestion de tâches et processus.

créés et identifiés selon le modèle général décrit précédemment (4.3.3). Ils sont décrits par un descripteur qui maintient une description de leur filiation ou de leur dépendance (processus et tâche support). A ceci s'ajoute un état d'exécution propre.

Graphe d'état des processus et tâches

Le graphe d'état d'un processus est simple (figure 4.14). Le graphe d'état d'une

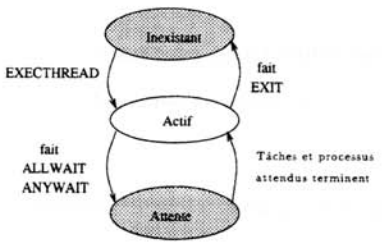

Figure 4.14: Graphe d'état des processus.

tâche est plus complexe du fait de son rôle de support d'exécution des processus

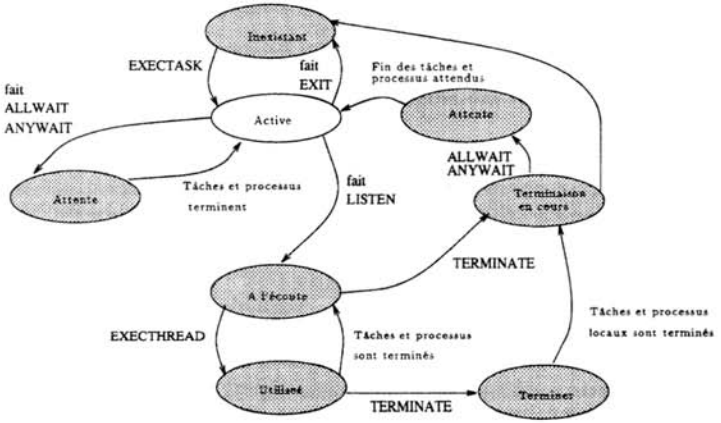

Figure 4.15: Graphe d'état des tâches.

(figure 4.15). Le graphe d'état matérialise les deux cas de fonctionnement d'une tâche. Le premier cas est identique à celui d'un processus qui naît, s'exécute et meurt, avec attente éventuelle de ses fils. Le second correspond à un rôle de serveur. Dans ce cas, la tâche est active indirectement comme support d'exécution de processus. Sa terminaison imposée de l'extérieur est différée jusqu'à terminaison des processus supportés. Dans sa séquence de terminaison, une tâche ne peut que proposer des indications de terminaisons (TERMINATE), de destruction d'entités (portes, barrières, etc.) ou d'attente de processus/tâches filles.

Descripteur d'une tâche/processus

Le descripteur d'un processus comprend :

- son état,
- l'identité du créateur (tâche ou processus),
- l'identité de la tâche support (locale),
- la liste d'attente,
- le mode d'attente.

La **liste d'attente** est la liste des identités des tâches et processus attendus. Le descripteur d'une tâche est identique, mais comprend en autre :

- la table des points d'entrée (donnée par LISTEN),
- la liste des processus en cours d'exécution sur la tâche,
- la liste des requêtes EXECTHREAD en attente.

Il est alors possible de décrire rapidement les fonctions des procédures de service.

Les procédures de service

Nous décrirons tout d'abord les procédures ne mettant en jeu que les serveurs sur lequel l'entité concernée (tâche ou processus) se trouve.

do_exectask (dest, path, tid, n, taille1, arg1, taille2, arg2, ...,taillen, argn) : l'objet tâche est créé, son identité forgée. La tâche doit être effectivement construite :

- allocation de la mémoire,
- chargement du code de la tâche,
- initialisation du descripteur,
- création du processus initial.

Les deux premiers points sont des opérations du Micro Noyau local au processeur. Le chargement n'est pas une action simple. Il met en cause une procédure de chargement-translation d'un fichier binaire objet. Le chargeur-translateur est une composante de notre environnement de programmation C et C//. L'opération de chargement nécessite l'accès à un fichier (la tâche est définie par un "chemin d'accès" à la Unix). Un serveur de fichier (et d'entrée sortie) fait parti des serveurs de base attaché à notre Micro Noyau. Ce serveur est placé sur une machine quelconque accessible de notre machine parallèle. Le chapitre 5 décrit l'environnement d'exploitation de la machine parallèle et l'organisation des accès à un serveur externe. Naturellement toute erreur dans cette séquence de création provoque l'abandon de la requête.

do_listen (n, "p1", "p2", ..., "pn") : accroche au descripteur de la tâche la table décrivant les points d'entrée exportés. Le processus initial de la tâche se suspend. Nous trouvons ici notre premier problème lié au parallélisme. En effet, après la création de la tâche, le processus créateur peut lancer une série de requêtes de création de processus sur la tâche (EXECTHREAD). La tâche lancée n'a pas nécessairement terminé son initialisation du fait d'installation de sous tâches ou processus. Ces requêtes doivent être différées jusqu'au LISTEN. Pour ce faire, il suffit de suspendre les processus esclaves (EXECTHREAD) tant que l'état de la tâche n'est pas à l'ÉCOUTE.

do_execthread (identificateur ,tleger-nom, tleger-id, n, taille1, arg1, taille2, arg2, ..., taillen, argn) : la création est différée tant que l'état de la tâche est ACTIF ou en ATTENTE. Dès que l'état est LISTEN, l'objet processus est créé, son identité forgée. Sa construction effective comprend :

- allocation de la mémoire pour la pile et initialisation (paramètres),
- initialisation du descripteur,
- mis à jour du descripteur de la tâche support,
- création du processus.

Si l'état de la tâche passe à INEXISTANT, les créations en attente sont avortées.

do_terminate (liste-id) : si la tâche est dans l'état à l'ÉCOUTE, le processus initial est reativé. Si la tâche est UTILISÉE, cette réactivation est différée jusqu'à ce que tous les processus supportés soient terminés. L'état de la tâche passe à A TERMINER pour interdire toute création ultérieure de processus.

do_status (processus-id) : un test sur une tâche/processus est effectué. Dans un premier temps, on teste l'existence de la tâche/processus. Trois cas sont reconnus selon les dates portées par l'identificateur de la tâche/processus et le descripteur obtenu à partir de l'indice extrait de l'identificateur unique :

- **identificateur.date < descripteur.date** : la tâche ou le processus sont déjà terminés,

- **identificateur.date > descripteur.date** : tâche ou processus inexistant,
- **identificateur.date = descripteur.date** : l'entité désignée existe.

Les procédures de service mettant en jeu plusieurs serveurs de gestion de tâches/-processus sont les procédures d'attente de terminaison de tâches et processus. Notre implantation utilise au mieux les propriétés de notre modèle, c'est à dire que seuls les créateurs peuvent attendre leusr enfants. Il s'en suit qu'il n'est pas nécessaire par processus ou tâche de gérer une liste de processus ou tâche en attente, car seul le créateur peut attendre. Nous avons choisi de prévenir systématiquement le créateur de la terminaison de ses enfants plutôt que d'établir un protocole de synchronisation spécifique entre requête d'attente et requête de terminaison.

do_exit : détruit le processus ou tâche courante (action locale). Le créateur dont l'identité est trouvé dans le descripteur à libérer reçoit la notification de la fin du processus ou tâche courante. Pour ce faire, une procédure supplémentaire est ajoutée au service de gestion de tâches/processus. Ce service est défini par une procédure stub classique :

```
SIGNAL_EXIT (identificateur_createur, identificateur_defunt) ;
```

qui est utilisée par le processus esclave effectuant **do_exit**. La procédure stub émet une requête à destination du site support du créateur qui exécutera la procédure de service **do_signal_exit**.

do_signal_exit : à la réception d'un SIGNAL_EXIT plusieurs cas sont possibles :

- le créateur n'existe plus : l'indication de terminaison est ignorée,
- le créateur n'est pas en attente de la tâche ou processus terminé : l'indication de terminaison est ignorée,
- le créateur est en attente de terminaison d'un ensemble de tâches/processus auquel appartient l'entité terminée : si le mode d'attente est ANY, le créateur est reactivé (c'est à dire, le processus esclave exécutent le **do_anywait** est réveillé. Si le mode est ALL, l'entité terminé est ôtée de la liste d'attente et le créateur n'est réveillé que si la liste est vide.

do_anywait, do_allwait : du fait que l'indication de terminaison d'un fils peut arriver alors que le créateur n'est pas à l'écoute et que ce signal est perdu, il

est nécessaire d'interroger les sites supports des entités pour savoir si les entités existent toujours. Ceci se fait simplement par une requête de type STATUS. Si aucune terminaison n'est trouvée par un ANYWAIT, le processus esclave se met en attente (état EN ATTENTE). La liste d'attente est attachée au descripteur de l'entité tâche/processus faisant le ANYWAIT. Dans le cas du ALLWAIT, tous les éléments de la liste doivent être terminés. La liste d'attente attachée au descripteur ne contient que les entités non terminées. L'inconvénient de cette implantation simple se trouve dans l'émission redondante de messages d'interrogation de terminaison nécessaires du fait du choix de ne pas gérer explicitement la filiation d'une entité tâche ou processus.

Remarque : le traitement du ALLWAIT/ANYWAIT fait apparaître l'interrogation de tous les sites support des entités de la liste. C'est en fait une diffusion de la requête STATUS sur toutes les entités de la liste. L'interrogation n'est pas faite entité après entité par le processus faisant le **do_allwait** ou **do_anywait**, mais via un service de diffusion que nous décrirons plus loin (4.3.7).

4.3.6 Le service de communication

Le service de communication gère deux types d'entités :

- le porte **Data**,
- et le porte **Message**.

Ces entités suivent le schéma général de gestion et d'identification mais diffèrent par leur sémantique.

Les portes Data

Les opérations possibles sont présentées dans la figure 4.16. Nous avons choisi une implantation simple où la porte **Data** est placée sur un site, ne migre pas et n'a pas de copies totales ou partielles sur d'autres sites. Le seul problème qui se pose est de définir la synchronisation entre les différents opérateurs pouvant être invoqués en parallèle. La solution choisie est simple et classique. C'est un protocole **lecteur/rédacteur** respectant strictement l'ordre FIFO. Les seuls opérateurs exécutables de façon concurrente sont les opérations GET_DATA (lecture). Toutes les

MESSAGE	PARAMETRES D'APPEL	REPONSE
PORT_DATA_CREATE	processeur, port-data-id	port-data-id
PORTE_DATA_DESTROY	port-data-id	0 si OK
GET_AND_LOCK	port-data-id, &data, sizeof(data)	data
PUT_AND_UNLOCK	port-data-id, &data, sizeof(data)	0 si OK
GET_DATA	port-data-id, &data, sizeof(data)	data
PUT_DATA	port-data-id, &data, sizeof(data)	0 si OK

Figure 4.16: Les opérations sur les portes Data.

autres opérations doivent être exécutées de façon exclusive (assimilées à une écriture). Le GET_AND_LOCK joue un rôle particulier. Outre une lecture exclusive, il verrouille la porte **Data** jusqu'à recevoir un PUT_AND_UNLOCK. Il s'ensuit que le PUT_AND_UNLOCK est le seul opérateur violant l'ordre FIFO (dans le cas contraire, il ne serait pas possible de débloquer l'accès à une porte **Data**). Le graphe d'état d'une porte **Data** est montré dans la figure 4.17. La destruction

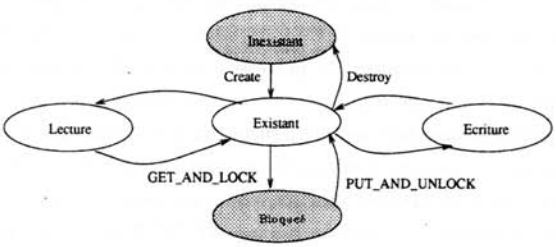

Figure 4.17: Graphe d'état des portes Data.

est considérée comme un opérateur normal (ordre FIFO) et non prioritaire qui détruit simplement la porte **Data**. Les requêtes en attente traitent cette situation comme un accès à un objet inexistant. L'utilisation d'une liste dans l'opération PUT_DATA est traitée comme une diffusion de l'écriture d'une même valeur sur les portes **Data** de la liste (cf. 4.3.7). L'implantation ne pose aucun problème du fait qu'on est ramené à synchroniser les seuls processus esclaves d'un unique serveur : celui du site possédant la porte. On adapte simplement un algorithme **lecteur/rédacteur** équitable (FIFO) classique écrit en termes de sémaphores.

Les portes Message

Les portes **Message** sont des dispositifs de communication analogues à des boites aux lettres sur lesquels on impose un ordre FIFO. L'ordre d'insertion définit l'ordre de retrait. Les opérateurs de communication sont présentés dans la figure 4.18. Notre choix d'implantation est simple. Une porte **Message** est une suite FIFO de

MESSAGE	PARAMETRES D'APPEL	REPONSE
PORT_MESSAGE_CREATE	processeur , &port-id	port-id
PORT_MESSAGE_DESTROY	port-id	0 si OK
SEND	port-id, m, sizeof(m)	0 si OK
SENDB	port-id, m, sizeof(m)	0 si OK
RECV	port-id, &m, sizeof(m)	m
RECVNB	port-id, &m, sizeof(m)	m si OK,1 si porte vide

Figure 4.18: Les opérateurs de communication.

messages enregistrés sur le site où la porte **Message** a été créée. L'implantation serait donc particulièrement triviale si l'on avait que des opérateurs de communication sur une porte à la fois. Le **do_send** n'est qu'un dépôt du message dans la liste avec réveil éventuel du processus esclave bloqué sur un **do_receive**. Le **do_sendb** nécessite que le processus émetteur attende la réception du message. Le processus esclave doit donc être suspendu jusqu'à **do_recv** prenant le message dans la file de message. Les opérations peuvent porter sur des listes de porte **Message**. SEND/SENDB est une diffusion d'un message à toutes les portes de la liste. RECV est une attente d'un message RECVNB sur une des portes de la liste. Le cas d'une liste de diffusion se traduit par une suite d'émission du message vers chacune des portes à la charge des serveurs de diffusion (4.3.7) et n'a pas d'incidence sur l'implantation du serveur de communication.

Par contre, la réception avec choix d'une porte parmi les portes prêtes à donner un message est l'implantation d'un opérateur impliquant une coopération entre plusieurs serveurs de communication :

- celui du site effectuant la requête RECV/RECVNB,
- ceux des sites portant les portes interrogées.

Le principe du protocole est simple. Le serveur de communication initiateur de la communication diffuse aux autres serveurs "demande de réservation" RESERV de

message. Lorsqu'une porte est prête à satisfaire la réservation, elle répond qu'elle "accepte la réservation" ACCEPT. Le serveur initiateur confirme CONFIRM ou annule ANNUL cette réservation. Il doit confirmer une réservation et annuler les autres (sémantique du RECV). Le protocole doit en outre :

- garantir l'ordre FIFO sur une porte Message,
- définir une règle de choix parmi les portes prêtes,
- distinguer différentes réservations sur les ensembles de portes identiques ou non.

Le premier point est facile à garantir dès que requêtes de réception ou de réservation sont traitées selon l'ordre FIFO. Le choix le plus évident est de prendre la première porte acceptant la réservation. Le dernier point nécessite que le serveur initiateur identifie une négociation de réservation et son état. Pour ce faire, une entité de communication supplémentaire est définie. Son usage est restreint au serveurs de communication. Elle est identifiée de façon unique par la méthode **<numéro du site, numéro local, date>** et possède un descripteur contenant :

- la liste des portes Message interrogées,
- l'état de la négociation ENCOURS/TERMINÉ,
- la porte sélectionnée (NIL au départ).

Les requêtes de manipulation de ces **"sélecteurs"** sont des requêtes de service propres aux serveurs de communication :

```
RESERV (porte_message_id, selecteur_id) ;
         ou
RESERV (liste_porte_message_id, selecteur_id) ;
ACCEPT (selecteur_id, porte_message_id) ;
CONFIRM (porte_message_id, selecteur_id, &message) ;
                                          |
                                          contient le RECV
ANNUL (porte_message_id, selecteur_id) ;
```

Le fonctionnement est alors le suivant :

Site initiateur

La procédure stub RECV/RECVNB détecte une situation simple (une porte) et émet sa requête vers le serveur possédant la porte. Dans le cas contraire, elle transmet la requête au serveur de communication local. Celui-ci (le processus esclave) effectue les actions suivantes :

do_recv (cas d'une liste) :

- crée un sélecteur et l'initialise,
- diffuse (via le serveur de diffusion) une requête RESERV,
- se met en attente de l'état TERMINÉ du sélecteur,
- à son réveil il émet une requête CONFIRM à destination de la porte sélectionnée afin d'acquérir le message pour le processus client local.

do_accept : le processus esclave retrouve le sélecteur, s'il existe. Le cas contraire indique que la sélection a été faite et qu'un ANNUL a croisé le ACCEPT. Rien n'est à faire. Si le descripteur est trouvé, l'état passe à TERMINE, et la porte sélectionnée est enregistrée. Le processus esclave en attente de la transition d'état à TERMINÉ est réveillé. Une ANNULATION est diffusée sur toutes les portes de la liste, sauf la porte sélectionnée.

Sites possédant les portes candidates

do_reserv : le processus esclave attend son tour pour acquérir un message. A son réveil, au lieu de retirer un message (cas du RECV sur une porte), il bloque la porte et émet une acceptation de la réservation. Il connaît le serveur initiateur à partir de l'identité du sélecteur. Il termine.

do_termine : le processus esclave retire le message et débloque la porte.

do_annul : le processus esclave peut se trouver dans deux cas :

- la porte est bloquée par le sélecteur figurant dans l'annulation. On débloque simplement la porte,

- la porte n'est pas bloquée par le sélecteur annulé. Une réservation est nécessairement en attente dans la file des requêtes de réservation ou de réception. Celle-ci est retirée.

4.3.7 Le service de diffusion

Nous avons fait plusieurs fois référence au besoin de diffuser un message, un opérateur, de façon identique, sur plusieurs objets. Le travail de diffusion est assuré par le serveur de diffusion. Nous avons considéré que l'utilisation d'un protocole de diffusion à un niveau logique (diffuser un opérateur sur un ensemble d'objets) était d'une part suffisamment spécifique, d'autre part suffisamment dépendante de la couche de communication pour être mise en oeuvre de façon spécifique. Le service de diffusion procède de la façon suivante : à la réception d'une requête de diffusion, le serveur initiateur procède à une partition de la liste des entités cibles de la diffusion en autant de listes qu'il y a des sites concernés. Il émet à destination des serveurs de diffusion de chacun des sites une requête de diffusion locale. C'est à dire que la liste ne contient que des entités locales au site. Chaque serveur assure alors la diffusion locale, c'est à dire, contacte le serveur responsable de l'objet pour lui faire effectuer les opérations requises.

En l'absence de protocole de communication offrant un mécanisme de diffusion, l'implantation proposé est la suivante :

- le serveur de diffusion initiateur émet les requêtes de diffusion locales successivement à tous les serveurs des sites concernés,
- chaque serveur de diffusion invoque le serveur concerné pour chacun des objets de la liste.

Une variante simple serait de créer autant de processus esclaves pour appeler ce serveur à condition que les opérations soient exécutables plus efficacement de façon concurrente.

Un inconvénient de la réalisation de la diffusion comme une suite de communications synchrone vers les autres serveurs est de ne pouvoir profiter du parallélisme interne au réseau de communication. Là aussi, une accélération peut être obtenue

de façon simple, en créant autant de processus esclaves que de sites concernés par la diffusion. On notera qu'une telle implantation du service de diffusion n'est possible que parce que notre environnement d'exécution n'impose aucune contrainte d'ordre particulier pour la diffusion (ordre total, ordre causal). Le seul ordre requis pour la communication est l'ordre FIFO. Celui-ci est garanti d'une part par le protocole RPC utilisé mais encore par la couche de communication du Micro Noyau.

4.3.8 Bilan

Nous avons proposé une implantation d'un environnement d'exécution parallèle comme :

- un Micro Noyau Parallèle réduit à un protocole RPC simple,
- un ensemble de serveurs coopérants à l'implantation des diverses entités de l'environnement parallèle.

Il est claire que le Micro Noyau proposé s'est révèle suffisamment puissant et souple pour supporter les choix d'implantation pour les serveurs nécessaires à la réalisation de cet environnement. Une lacune évidente de ce noyau est l'absence de protocole de diffusion efficace exploitant mieux les propriétés du réseau d'interconnexion. Il faut reconnaître à notre décharge que de proposer un protocole de diffusion efficace du type arbre couvrant dans le contexte d'une machine reconfigurable est encore un problème de recherche. Pour ce faire, il aurait fallu travailler dans le cadre d'une géométrie de réseau statique : une grille 2D dans le cadre du Meganode.

Les autres limitations concernent plus directement les choix d'implantation de notre environnement de programmation. Elles concernent plus particulièrement l'implantation des portes Data et Message. Dans les portes Data, nous avons choisi une implantation centralisée au risque de créer un goulot d'étranglement et au prix d'une inhibition du parallélisme possible entre lectures. Une possibilité aurait été d'autoriser des copies complètes ou partielles (cache). Cette parallélisation des accès aurait nécessité la mise en place d'un protocole de maintien de cohérence entre ces copies. Tous les protocoles existants exigent la mise en place

d'un mécanisme de diffusion **atomique ou causal**[Jos89]) dont le coût aurait été élevé dans notre cas.

En ce qui concerne les portes Message, le choix d'un concept boite à lettre par rapport à celui de flot bi-point, nous interdit de tirer partie de la bande passante du réseau en utilisant des techniques de fenêtre d'émission-fenêtre de réception. Ces choix limitatifs nous ont cependant permis de mener à bien l'étude, la définition et la réalisation d'une partie importante d'un environnement de programmation parallèle. A l'heure actuelle, le Micro Noyau Parallèle est implanté, ainsi qu'un environnement d'exploitation permet de communiquer avec un réseau de stations (cf. chapitre 5). Les serveurs de gestion de tâches/processus et de communication sont partiellement implantés. Un travail important de finition et test reste à faire.

Chapitre 5

L'environnement de programmation

5.1 Introduction

Nous n'avons pas de facilités spécifiques pour la **production de programmes** C// ni pour leur **évaluation**. L'utilisateur écrit les programmes C// sur une machine Unix en utilisant le compilateur C// et le Micro Noyau Parallèle assure son exécution. Ainsi, notre environnement de programmation se résume à **l'environnement d'exploitation** de la machine Supernode, que nous présentons par la suite.

La plupart des machines parallèles sont utilisées via une machine hôte qui offre des services **d'accès à la machine parallèle**, de **chargement initial du programme parallèle** et des fonctions **d'entrée/sortie** et de **gestion des fichiers**. Un moyen d'étendre les possibilités d'utilisation de la machine parallèle est de la connecter à un ordinateur hôte lui même connecté à un réseau. Ainsi, tous les utilisateurs sur des machines du réseau qui veulent exécuter des programmes parallèles demandent l'accès à la machine, qui est gérée comme une ressource commune. Dans les sections suivantes, nous allons présenter la solution que nous avons adopté au sein du groupe PLoSys pour l'exploitation de la machine parallèle Supernode. Cette solution a pour principale caractéristique la simplicité et la rapidité de réalisation. Notre environnement se réduit à :

- l'utilisation d'une passerelle Supernode/réseau de faible coût (micro ordinateur PC),
- un serveur d'entrée/sortie et d'accès aux fichiers Unix appelé de façon transparente depuis les programmes parallèles, et
- un chargeur initial qui permet le chargement du Micro Noyau sur tous les Transputers.

5.2 L'environnement matériel et la configuration initiale

La machine Supernode a été conçue pour être utilisée avec un ordinateur frontal. La communication entre la machine hôte et la machine Supernode doit se faire à partir d'une carte Transputer insérée dans la machine hôte, car le Supernode ne possède pas de bus standard. Un micro ordinateur PC équipé d'une carte Ethernet et d'une carte Transputer joue le rôle de passerelle d'accès entre stations connectés au réseau et le Supernode. La figure 5.1 montre cette organisation. La

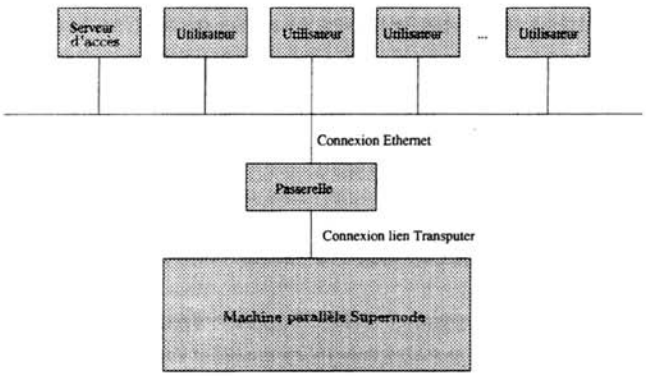

Figure 5.1: Organisation générale de l'environnement matériel.

carte comportant un Transputer est placée dans le micro ordinateur PC. Un des liens de ce Transputer est connecté à un circuit de conversion série/parallèle

réalisant l'interface entre ce Transputer et l'ordinateur hôte, via une interface bus standard. Le Transputer comportant quatre liens physiques, trois sont donc utilisables pour communiquer avec les Transputers du Supernode. Un exemple de connexion est montré dans la figure 5.2. Dans cette figure, un lien de la carte interface est connecté à un lien du Transputer de contrôle. Dans une configuration en anneau du Supernode, le Transputer de la carte d'interface peut être inséré dans l'anneau ce qui permet les opérations d'entrée/sortie et d'accès aux fichiers sur les machines Unix du réseau local. La gestion des accès au Supernode comme les services d'entrée/sortie et gestion de fichiers qui lui sont fournis sont assurés par une station Unix dit serveur d'accès.

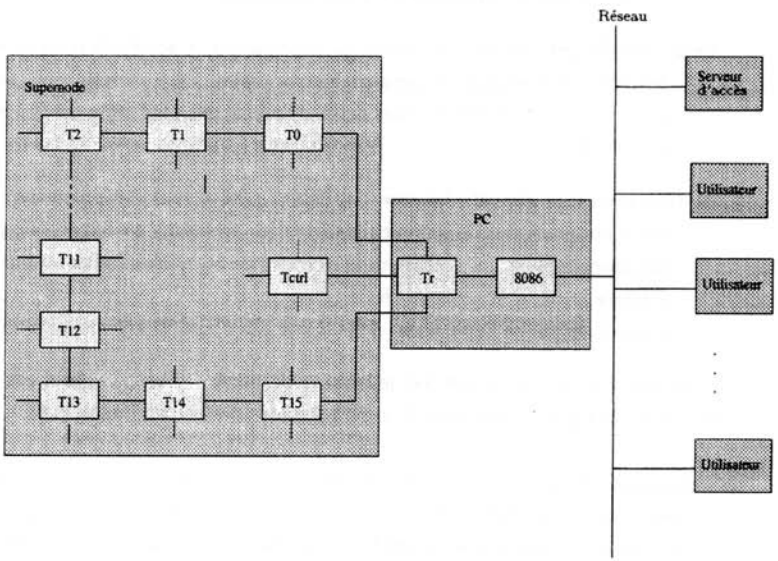

Figure 5.2: La machine Supernode configurée en anneau.

5.3 Le chargement

Le système est organisé autour de trois entités logiques : la **passerelle**, le **serveur d'accès** et le **serveur d'usage général**.

La passerelle : la passerelle assure les transferts de données entre la carte interface et le serveur d'accès. Comme processeurs et systèmes sont différents, il est nécessaire de convertir le protocole RPC de notre Micro Noyau parallèle dans un protocole acceptable pour les machines des utilisateurs. Ainsi, la **passerelle** prend un message d'appel RPC qui arrive de la machine Supernode (protocole Transputer), le transforme dans le format de la machine de l'**utilisateur**, l'encapsule dans un paquet IP, l'envoie au **serveur d'accès** qui le redistribue aux processus utilisateurs. Pour la REPONSE, la **passerelle** extrait du paquet IP le message, le transforme dans le format Transputer, et l'envoie sur le **Transputer de la carte interface**.

Le serveur d'accès : en plus du travail de distribution, le **serveur d'accès** gère les demandes d'exécution de programme parallèles. Cette fonction est nécessaire du fait la concurrence des accès possibles dans un contexte de réseau local Unix.

Le serveur d'usage général : les **serveurs d'usage général** sont des processus Unix communiquant avec le serveur d'accès et les processus du Supernode pour initialiser le Supernode, charger un programme parallèle et offrir les services d'entrée/sortie et d'accès aux fichiers aux processus du programme parallèle s'exécutant sur le Supernode, pour le compte d'un utilisateur particulier.

Le **serveur d'accès** est unique par le réseau. Un **serveur d'usage général** est créé spécifiquement pour l'exécution d'un programme parallèle d'un utilisateur.

Cette organisation nécessite l'établissement de circuits virtuels pour l'échange de données entre ces entités. Chaque entité logique est réalisée par un programme C//. Il existe un programme **passerelle** qui tourne sur le micro ordinateur PC, un programme **serveur d'accès** qui tourne sur une machine Unix et un programme **serveur d'usage général** sur chaque machine Unix du réseau qui accède au Supernode.

Pour le **chargement initial** du système, on connecte logiquement la **passerelle** et le **serveur d'accès**, puis le **serveur d'accès** transfére le Micro Noyau sur la **passerelle**. La **passerelle** charge le Micro Noyau sur le **Transputer de la carte interface**. Ce Micro Noyau possède le serveur **gestionnaire de tâches**. Ce serveur s'initialise et exécute (lui même) une opération EXECTASK pour démarrer un programme appelé INIT.

Le programme INIT alors est chargé sur le **Transputer de la carte interface** et démarre. Ce programme exécute successivement les opérations de chargement local des différents serveurs (communication, synchronisation). Ensuite, il charge le Micro Noyau sur le Transputer de contrôle et exécute des opérations distantes pour charger les mêmes serveurs. Il lance aussi deux serveurs spéciaux sur le Transputer de contrôle : le **gestionnaire de contrôle** et le **gestionnaire de connections**. Par des appels RPC au **gestionnaire de connections**, le programme INIT configure le Supernode en anneau. Puis, il charge le Micro Noyau sur un des Transputers de l'anneau connecté au Transputer de la carte interface (où tourne INIT). Le Micro Noyau démarre sur ce Transputer, charge les serveurs et charge, lui aussi, son voisin. Ainsi de suite, chaque Transputer charge le Micro Noyau sur son voisin. Une fois terminée la phase de chargement initiale, le Micro Noyau est installé sur tous les Transputers et les serveurs attendent des requêtes. Le programme INIT se comporte alors comme un petit "**shell**" qui attend les commandes de l'utilisateur, en particulier celle de lancement d'un programme parallèle. De son côte, le **serveur d'accès**, après avoir charge le Micro Noyau, attend une connexion d'un utilisateur pour l'exécution d'un programme parallèle. A la connexion, le **serveur d'usage général** est lancé.

5.4 L'accès aux services externes

Un programme parallèle qui tourne sur la machine Supernode est composé par des processus qui exécutent des opérations d'entrée/sortie et d'accès aux fichiers, qui sont assurées par un serveur dit **d'usage général**. C'est à dire qu'il n'existe pas une occurrence de ce serveur sur le Micro Noyau Parallèle qui tourne sur les Transputers (machine Supernode et carte interface). Pour adresser une requête à ce serveur, le principe est le même que celui utilisé pour les autres serveurs du système : un client sur un Transputer génère un appel RPC qui est transformé par la procédure **stub** correspondante en un message pour le serveur concerné. Ce message contient l'identification du **processeur/serveur** destinataire, **identification de la fonction** et les **paramètres**. Ce message est donc acheminé via les tables de routage jusqu'au Transputer Interface qui l'envoie sur la passerelle. La **passerelle** assure la conversion de format du message (Transputer/machine Unix) le met dans un paquet IP et l'envoie au **serveur d'accès**, que le redirige sur le **serveur**

d'usage général. Celui-ci analyse le message, identifie le service, et appelle la procédure qui l'exécute. La réponse est envoyée au serveur d'accès qui la retransmet à la passerelle. La **passerelle** extrait le message du paquet IP, fait la conversion du format machine Unix en format Transputer et envoie le message sur le **Transputer de la carte interface**. A partir de cela, le message de REPONSE sera envoie au Transputer concerné, avec l'utilisation de la fonction de **routage**.

5.5 Conclusion

Nous avons présenté une version simple de l'environnement d'exploitation de la machine Supernode, que nous avons implanté. Nous envisageons d'implanter une version plus complète avec un **interpréteur de commandes** et un **gestionnaire de programmes**. Dans cette version plus complète, l'utilisateur sur une machine du réseau démarre un programme **utilisateur** qui demande une connexion avec le **serveur d'accès**. Celui-ci accepte la liaison et crée un processus fils qui exécute l'**interpréteur** de commandes. Ainsi, chaque utilisateur a un processus propre **interpréteur de commandes** qui reçoit les commandes (via les sockets) tapés sur le clavier, et les transmet au **gestionnaire de programmes** qui les ajoute dans la file de programmes à exécuter.

Le **gestionnaire de programmes** assure l'enchaînement de l'exécution des programmes sur la machine parallèle. Il gère la file de programmes à exécuter et fait le placement initial.

Chapitre 6

Conclusion

6.1 Bilan

Dans cette thèse, nous avons proposé un **environnement d'exécution parallèle** pour des machines du type NORMA. Cet environnement est la concrétisation d'un modèle de programme. Nous en proposons une réalisation utilisant systématiquement un modèle d'organisation client/serveur. Ce modèle est concrétisé par un Micro Noyau Parallèle très simple et puissant basé sur un mécanisme élémentaire de RPC.

Le modèle parallèle

Dans notre étude, nous avons distingué trois modèles de programme pour machines NORMA :

- processus communicants,
- client/serveur,
- mémoire partagée.

Dans le modèle de **processus communicants**, les processus communiquent explicitement par des opérations du type SEND et RECV. Dans le modèle **client/serveur**, les processus clients demandent des services aux serveurs via un mécanisme d'appel de procédure à distance (RPC). Le modèle à **mémoire partagée** présenté a été le modèle Linda. Dans ce modèle, les processus communiquent explicitement et

les données peuvent être distribuées sur les différents processeurs de la machine. Le modèle client/serveur est plus facile à utiliser, car plus classique que le modèle de processus communicants. Par contre, le modèle de processus communicants est plus performant. Le modèle à mémoire partagée nécessite un mécanisme pour maintenir la cohérence des données, car la parallélisation est assurée par l'existence de de copies multiples.

Le modèle de programme que nous avons proposé est basé sur l'échange de messages, et permet la programmation de différents paradigmes de programmation. Les principales caractéristiques de notre modèle sont :

- le concept de porte message est un modèle de processus communicants,
- les barrières permettent la synchronisation des processus sur différents processeurs,
- il supporte le modèle client/serveur asynchrone, par la possibilité de créer des processus à distance,
- il supporte une forme restreinte de mémoire partagée contrôlée par le programmeur : la porte Data,
- il supporte la création dynamique de processus, portes et barrières localement ou à distance.

Bien que nous n'ayons pas évalué systématiquement notre modèle, nous pensons qu'il est un compromis simple des différents modèles étudies.

Architecture

L'étude de l'implantation de notre modèle avait pour objectif de mettre en évidence une méthode de construction régulière. Nous avons choisi une architecture client/serveur où le principe de réalisation des services de gestion de tâches/-processus, de synchronisation, et communication est systématique. Les serveurs sont répliqués sur tous les sites et la localisation d'un serveur est assurée soit directement par la procédure stub du client, soit par le serveur local qui assure alors le rôle de coordinateur des autres serveurs impliqués dans le service. Cette méthode

est concrétisée par la définition d'un Micro Noyau Parallèle et son implantation sur la machine Supernode. Compte tenu d'un choix de réalisation simple de la plupart des services implantant notre modèle de communication, nous pensons que ce Micro Noyau est suffisant. Une lacune serait l'absence de protocole de diffusion associé au protocole RPC et exploitant la géométrie de la machine. On notera cependant qu'un protocole de diffusion efficace pour une géométrie changeant dynamiquement est un thème de recherche spécifique. L'implantation des différents fonctions de notre modèle à suivi une règle de simplicité et n'a pas présenté des difficultés particulières du fait du peu de contraintes imposées par le modèle.

La réalisation

La réalisation effectuée sur le Supernode comprend :

- le Micro Noyau Parallèle,
- les serveurs,
- un environnement permettant l'accès de/vers un réseau local Unix.

Ma contribution personnelle à cette réalisation à porté sur la définition et réalisation des serveurs, ainsi que l'environnement d'exploitation. Une grande partie du Micro Noyau et des outils (compilateur C//, etc.) est hérité du projet OPERA.

6.2 Perspectives

Un travail à court terme serait d'achever le prototype et d'évaluer l'environnement d'exécution proposé afin de vérifier son adéquation à la programmation parallèle ou les points de l'implantation critiques. Parmi ceux-ci, la fonction de communication est le point le plus important pour les performances d'un système parallèle. Le premier point susceptible d'être amélioré sont les **actions pluralistes**. Les **actions pluralistes** sont importantes dans un système parallèle car elles offrent aux utilisateurs la possibilité d'envoyer un message à plusieurs récepteurs, écrire dans plusieurs portes données, et placer plusieurs processus sur des sites différents. Cependant les opérations pluralistes proposées ne sont pas atomiques. De

ce fait, on ne peut pas assurer la cohérence d'actions parallèles sans synchronisation additive. Cependant une diffusion atomique ou causale reste coûteuse.

Une autre amélioration concerne les portes qui, situées nécessairement sur un site, risquent de crér un goulot d'étranglement et sérialisent les accès. La parallélisation des accès aux portes Data pourrait se faire par une technique de copie ou de cache (duplication partielle), ce qui imposerait la mise en place d'un protocole de maintien de cohérence. En ce qui concerne les portes Message, le problème est plus complexe et peut être eut il fallu associer une porte à une tâche consommatrice permettant ainsi de mettre en place des mécanismes simple du type fenêtre émission/fenêtre réception.

Ces points ne sont que des cas particuliers du problème général de l'équilibrage de charge des éléments de la machine parallèle. Le placement des portes conditionne la charge de communication de même que celui des processus conditionne la charge des noeuds de calcul. La migration d'un processus est un moyen de contrôler la charge de calcul comme celle de communication.

On peut se demander aussi si le parallélisme devenant un outil commun, il ne faut pas étendre le concept de machine parallèle à l'ensemble des moyens de calcul existant sur un site géographique donné. Dans ce cas, l'hétérogénéité des matériels deviendrait dominante et le problème d'un environnement d'exécution parallèle portable et hétérogène serait le problème à résoudre.

Pour conclure, l'expérience que nous a apporté le développement de cette thèse, plus précisément, la programmation du Micro Noyau Parallèle, nous a montré la difficulté de travailler sur des machines parallèles NORMA. Les phases de débogage et test demandent un effort qui ne peut se mesurer a priori, du fait du manque d'environnement d'évaluation et mise au point. Nous pensons qu'il y a encore beaucoup de travail à faire dans le domaine du parallélisme avant que les machines NORMA soient accessibles par tous les types d'utilisateurs. Pour l'instant, elles sont destinées à un personnel spécialisé qui les utilise de manière rudimentaire.

Bibliographie

[Acc86] Acceta, M. J. et al. Mach: A New Kernel Foundation for UNIX Development. In *Proceedings of Summer Usenix*, July, 1986, pp. 93-112.

[All86] Allen, R., Baumgartner, D., Kennedy, K., Porterfield, A. PTOOL: A Semi-automatic Parallel Programming Assistant. In *Proc. 1986 Int. Conf. Parallel Processing, St Charles, IEEE Computer Society Press*, August 1986, pp. 164-170.

[Ana92] Ananda, A. L. and Tay, B. H. A survey of Asyncrhonous Remote Procedure Calls. In *ACM Operating Systems Review*, Vol. 26, No. 2, Avril 1992.

[And83] Andrews, Gregory R. and Schneider, Fred B. Concepts and Notations for Concurrent Programming. In *ACM Computing Surveys*, Vol. 15, No. 1, March 1983.

[And91] Andrews, Gregory R. Paradigms for Process Interaction in Distributed Programs. In *ACM Computing Surveys*, Vol. 23, No. 1, March 1991, pp. 49-90.

[App89] Appelbe, Bill and Smith, Kevin. Start/Pat: A parallel-Programming Toolkit. In *IEEE Software*, July 1989, pp. 29-38.

[Arm89] Armand, François et al. Distributing UNIX Brings to its Original Virtues. In *Chorus Systèmes*, August 1989.

[Bai88] Baillie, Clive F. Comparing shared and distributed memory computers. In *Parallel Computing 8*, 1988, pp. 101-110.

[Bal89] Bal, Henry E., Steiner, Jennifer G. and Tanembaum, A.S. Programming Languages for Distributed Computing Systems. In *ACM Computing Surveys*, Vol. 21, No. 3, September 1989, pp. 261-322.

[Bal90] Bal, Henry E. Programming Distributed Systems. In *Prentice Hall*, 1990.

[Bal92] Bal H.E., Kaashoek, M.F. and Tanembaum, A.S. Orca: A Language For Parallel Programming of Distributed Systems. In *IEEE Transactions on Software Engineering* , Vol. 18, No. 3, March 1992 pp. 190-205.

[Ball90] Ballegeer, J.C. Le système d'exploitation distribué Helios. In *La lettre du Transputer et des calculateurs distribués*, No. 5, March 1990, pp. 33-39.

[Bec89] Beck, Bob and Olien, Dave. A Parallel-Programming Process Model. In *IEEE Software*, May 1989, pp. 63-72.

[Bir84] Birrel, A. D. and Nelson, B. J. Implementing Remote Procedure Calls. In *ACM Trans. Computer Systems*, Vol. 2, No. 1, Feb. 1984, pp. 39-59.

[Bro89] Browne, J.C., Azam, M., and Sobeck, S. CODE: A unified Approach to Parallel Programming. In *IEEE Software*, July 1989, pp. 10-18.

[Bri90a] Briat, J., Favre, M. and Geyer, C. Opera: Ou Parallélisme et Régulation Adaptative en Prolog. In *Actes du Séminaire de Programmation en Logique Trégastel* Trégastel, 1990, pp. 329-350.

[Bri90b] Briat, J., Favre, M., Geyer, C. and Chassin de Kergommeaux, J. Opera: A Parallel Prolog System an its Implementation on Supernode. In *International Workshop on Compilers for Parallel Computers*, Paris, 1990.

[Bri91] Briat, J., Favre, M. and Chassin de Kergommeaux, J. Scheduling of OR-parallel Prolog on a Scalable, Reconfigurable, Distributed-Memory Multiprocessor. In *PARLE'91*, Vol. II, June 1991, pp. 385-402.

[Bri92] Briat, J., Favre, M. and Da costa, Celso Maciel. Un noyau de système d'exploitation pour une machine parallèle sans mémoire commune. In *4es Rencontres du Parallélisme*, Université des Sciences et Technologies de Lille, Villeneuve d'Asc, 18-20 Mars 1992.

[But92] Butler, Ralph and Lusk, Ewing. User's guide to the p4 Programming system. In *ANL-92/17 - Mathematics and Computer Science Division, Argonne National Laboratory*, October 1992.

[Cme89] Cmelik, Robert F., Gehani, N.H. and Roome, W.D. Experience with Multiple Processor Versions of Concurrent C. In *IEEE Transactions on Software Engineering*, Vol. 15, No 3, March 1989, pp. 335-344.

[Cal88] Callahan, C.David, Cooper, Keith D., Hood, Robert T., Kennedy, Ken and Torcson, Linda. Parascope: A parallel programming environnement. In *The International Journal of Supercomputer Applications, Vol. 2 No. 4* , 1988, pp. 84-99.

[Car87] Carle, A., Cooper, Keith D., Hood, Robert T., Kennedy, Ken, Torcson, Linda, and Scoot, K.Warren. A Practical Environnement for Scientific Programming. In *IEEE Computer*, november 1987, pp. 75-89.

[Car89] Carriero, Nicholas and Gelernter, David. How to write parallel programs: A Guide to Perplexed. In *ACM Computing Surveys*, Vol. 21, No. 3, September 1989.

[Car91] Carpenter, D.B. Spawning Process Networks in a Virtual Channel Environnement. In *Department of Eletronic and Computer Science, University of Southampton*, November 1991.

[Cher88] Cheriton, David R. The V Distributed System. In *Communications of the ACM*, vol. 31, no. 3, March 1988.

[Coo88] Cooper, Eric C. and Draves, Richard P. C Threads. Technical Report CMU-CS-88-154, Computer Science Department, Carnegie Mellon University, June 1988.

[Dal87] Dally, W.J. et Seitz, C.L. Deadlock-Free Message Routing in Multiprocessor Interconnection Network. In *IEEE Transactions on Computers*, Vol. C-36, No. 5, May 1987, pp. 547-553.

[Des89] Despoix, Frédéric. Noyau de Communication pour réseaux réconfigurable de transputers. In Rapport de Diplome d'Etudes Approfondies, Laboratorie

de Génie Informatique, Institut de Mathematiques Apliquées de Grenoble, Grenoble-France, 1989.

[Din89] Dinning, Anne. A Survey of Synchronization Methods for Parallel Computers. In *IEEE Computer*, July 1989, pp. 66-77.

[Fav89] Favre, Michel, Santana, Miguel. Le langage C//. In *Institut National Polytechnique de Grenoble, Laboratorie de Génie Informatique*, Rapport interne, Grenoble-France, 1989.

[Fav92] Favre, Michel. Un système Prolog Parallèle pour machine à mémoire distribuée. In *Institut National Polytechnique de Grenoble, Laboratorie de Génie Informatique*, Thèse de doctorat, Grenoble-France, 1992.

[Fly72] Flynn, Michel J. Some Computer Organizations and their Effectiveness. In *IEEE Transactions on Computers*, vol C-21, 1972.

[For89] Fort, Didier. Un noyau de communication pour machines parallèles. In Rapport de Diplome d'Etudes Approfondies, Laboratorie de Génie Informatique, Institut de Mathematiques Apliquées de Grenoble, Grenoble-France, 1989.

[Gar90] Garret, Paul. Abstract Machine for scientific computation. In *Supercomputing* , March 1990, pp. 37-44.

[Geh86] Gehani, N.H. and Roome, W.D. Concurrent C. In *IEEE Software-Practice and Experience* , Vol. 16(9), Septembre 1986, pp. 821-844.

[Geh89] Gehani, N.H. and Roome, W.D. The Concurrent C Programming Language. In *Prentice Hall*, 1989.

[Geh92] Gehani, N.H. and Roome, W.D. Implementing Concurrent C. In *IEEE Software-Practice and Experience* , Vol. 22(3), March 1992, pp. 265-285.

[Gel85] Gelernter, David. Generative Communication in Linda. In *ACM Transactions on Programming Languages and Systems*, Vol. 7, No. 1, January 1985, pp. 80-112.

[Gey91] Geyer, C. F. R. Une contribution à l'étude du parallélisme OU en Prolog sur des machines sans mémoire commune. In *Université Joseph Fourier, Labotatoire de Génie Informatique*, Thése de doctorat, Grenoble - France, 1991.

[Gon91] Gonzales Valenzuela, Néstor Alejandro. PARX: Noyau de système pour les ordinateurs massivement parallèles - Contrôle de la communication entre processus. In *Institut National Polytechnique de Grenoble, Laboratorie de Génie Informatique*, Thése de doctorat, Grenoble-France, 1991.

[Gua89] Guarna, Vincent A., Jr., Dennis Gannon, Jablonowski, David, Malony, Allen D. and Gaur, Yogesh. Faust: An integrated Environnement for Parallel Programming. In *IEEE Software*, July 1989, pp. 20-27.

[Hag92] Hagersten, Erik, L., Landin, A. and Haridi, S. DDM - A Cache-Only Memory Architecture. In *IEEE Computer*, September 1992, pp. 44-54.

[Hoa78] Hoare, C.A.R. Communicating Sequential Process. In *Communications of the ACM*, vol. 21, no. 8, August 1978.

[Int86] Inter-Process communication. Sun Microsystems, 1986.

[Jaz91] Jazayeri et al., P3M: An Abstract Architecture for Massively Parallel Machine. *Workshop on Abstract Machine Models for Highly Parallel Computers*, University of leeds, March 25-27, 1991, pp. 39-45.

[Jos89] Joseph, T. A. and Birman, K. P. Reliable Broadcast Protocols. In *Distributed System - Edited by Sape Mullender*, Addison-Wesley Publishing Company, 1989.

[Kar87] Karp, Alan H. Programming for parallelism. In *IEEE Computer*, May 1987, pp. 43-57.

[Ken91] Kennedy, K., Mckinley, Kathryn S., and Tseng, Chau-Wen. Interactive Parallel Programming Using the Parascope Editor. In *IEEE Transactions on Parallel and Distributed Systems*, Vol. 2, No. 3, July 1991, pp. 329-341.

[Laz91] Lazarov, V.,Iliev,R., Djendov, D. Migrating Dynamic Process in the Gamma Parallel Machine. In *Centre for Informatics and Computer Technology - Bulgarian Academy of Sciences*, Sofia - Bulgaria, 1991.

[Lew92] Lewis Ted G. Introduction To Parallel Computing. In *Prentice Hall International Editions*, 1992.

[Lis88] Liskov, Barbara. Promises: Linguistic Support for Efficient Asynchronous Procedure Calls in Distributed Systems. In *SIGPLAN*, Atlanta, Georgia, June 1988.

[Mac91] Macdonald, N. and Norman, M.G. Issues in Parallel Programming Environnements. *Workshop on Abstract Machine Models for Highly Parallel Computers*, University of leeds, March 25-27, 1991, pp. 23-27.

[Mag91] Magee, J. and Dulay, N. Configuring Parallel Programs. *Workshop on Abstract Machine Models for Highly Parallel Computers*, University of leeds, March 25-27, 1991, pp. 89-94.

[Mar88] Martins, J. Legatheaux and Bervers, Y. La désignation dans les systèmes d'exploitation répartis. In *T.S.I. - Technique et Science Informatique*, Vol. 7, No. 4, 1988, pp. 359-372.

[Mas89] Masini, Gérald, Napoli, A., Colnet, D., Léonard, D. et Tombre, K. Les langages à objets. In *Ed. InterEditions*, 1989.

[Mor90] Morris, D., Theaker, R., Phillips, R., Evans, D.G. An experimental parallel system (EPS). In *parallel Computing 15*, 1990, pp. 247-259.

[Muh88] Muhlenbein, H., Kramer, O., Limburger, F., Mevenkamp, M. and Streitz, S. MUPPET: A programming environnement for message-based multiprocessors. In *Parallel Computing 8*, 1988, pp. 201-221.

[Mul90] Mullender,Sape J. and Van Rossum, Guido. AMOEBA A Distributed Operating System For the 1990s. In *IEEE Computer*, May 1990, pp.44-53.

[Mun90] Muntean, Traian et Waille, Philippe. L'architecture des machines Supernode. In *La lettre du Transputer*, No. 7, Septembre 1990, pp. 11-40.

[New87] Newton, Thomas D. An Implementation of Ada Tasking. *Carnegie Mellon*, CMU-CS-87-169, October 1987.

[PC-NFS89] PC-NFS 3.0.1 Sun Microsystems, 1989.

[Pre91] Presburg, D., Wisneski J. et all. Code, an assembly language for an abstract SIMD Machine. *Workshop on Abstract Machine Models for Highly Parallel Computers*, University of leeds, March 25-27, 1991, pp. 59-64.

[Ray88] Raynal, Michel et Helary, Jean-Michel. Synchronisation et contrôle des systèmes et des programmes répartis. *Ed. Eyrolles*, 1988.

[Ray91] Raynal, Michel. La communication et le temps dans les réseaux et les sysèmes répartis. *Ed. Eyrolles*, 1991.

[Riv87] Riveill Michel. CONKER : un modèle de répartition pour processus communicants. Application à Occam. In *Institut National Polytechnique de Grenoble, Laboratorie de Génie Informatique*, Thèse de doctorat, Grenoble-France, 1987.

[Riz92] Rizzo, Luigi. PCserver: Networking PC-hosted Transputers. *Transputers'92*, Arc-et-Senans, France, 20-22 May 1992, pp. 158-171.

[Sat87] Satyanarayanan, M and Siegel, Helen H. MultiRPC: A Parallel Remote Procedure Call Mechanism. *Carnegie Mellon*, CMU-CS-87-136-A, May 1987.

[Sch91] Schneider, A. and King, A. Mona Lisa. A Paradigm for Coarse Grain Parallelisme. SIMD Machine. *Workshop on Abstract Machine Models for Highly Parallel Computers*, University of leeds, March 25-27, 1991, pp. 61-74.

[Sne88] Snelling, David F. and Hoffmann, Geerd-R. A comparative study of libraries for parallel processing. In *parallel Computing 8*, 1988, pp. 255-266.

[Ste90] Stevens, Richard W. Unix Network Programming. *Prentice-Hall*, 1990.

[Tan87] Tanembaum, Andrew S. Operating systems: Design and Implementation. In *Prentice Hall* , 1987.

[Tan90] Tanembaum, Andrew S. Réseau, Architectures, protocoles, applications. In *InterEditions*, 1990.

[Tan90a] Tanembaum, Andrew S. et al. Experiences with the Amoeba distributed system. In *Communications of the ACM*, vol. 33, no. 12, Dec 1990.

[Tan92] Tanembaum, Andrew S. Modern Operating Systems. In *Prentice Hall*, 1992.

[Tan92] Tanembaum, Andrew S., Kaashoek, M.Frans and Bal, Henri E. Parallel Programming Using Shared Objects and Broadcasting. In *IEEE Computer*, August 1992 pp. 10-19.

[Tho89] Thomas, Bernhard and Peinze, Klaus. Suprenum Confort of Parallel Programming. In *Supercomputing* , March 1989, pp. 31-43.

[Twi90] Twist, Rob Van. Report on the TROPICS abstract machine. Philips Research Laboratories Eindhoven, 1990.

[Tay90] Tay, B. H. and Ananda, A.L. A survey of Remote Procedure Calls. In *ACM Operating Systems Review*, Vol. 24, No. 3, July 1990.

[Vee91] Veer, B. The Helios Model of Networks. *Workshop on Abstract Machine Models for Highly Parallel Computers*, University of leeds, March 25-27, 1991, pp. 81-89.

[Wal90] Walker, Edward F., Floyd, Richard and Neves, Paul. Asynchronous Remote Operation in Execution Distributed Systems. In *The 10th Conference on Distributed Computing Systems*, Paris, France, May 1990.

[Wer89] Werner, Karl Heinz et all. The Suprenum User Interface. In *Supercomputing* , March 1989, pp. 20-24.

[Xu92] Xu, Hong, McKinley, Philip K. and Ni, Lionel M. Efficient Implementation of Barrier Synchronisation in Wormhole-Routed Hypercube Multicomputers. In *The 12th Conference on Distributed Computing Systems*, Yokohama, Japan, June 1992.

[Yan92] Yang, Cui-Qing. Distributed Computing In A NUMP (Non-Uniform Message Passing) Environnement. In *ACM Operating Systems Review*, Vol. 26, No. 2, Avril 1992.

Résumé

L'objectif de cette thèse est l'étude d'un environnement d'exécution pour machines parallèles sans mémoire commune. Elle comprend la définition d'un modèle de programme parallèle, basé sur l'échange de message offrant une forme restreinte de mémoire partagée. La communication est indirecte, via des portes; les processus utilisent les barrières pour la synchronisation. Les entités du système, processus, portes et barrières, sont créées dynamiquement, et placées sur un processeur quelconque du réseau de processeurs de façon explicite.

Nous proposons une implantation de ce modèle comme la mise en oeuvre systématique d'une architecture client/serveur. Cette implantation a été effectuée sur une machine Supernode. La base est un Micro Noyau Parallèle, où le composant principal est un mécanisme d'appel de procédure à distance minimal.

Mots clefs : Machines parallèles à mémoire distribuée, Modèles de programmes parallèles, Environnement d'exécution, Modèle client/serveur, Appel de procédure à distance, Micro Noyau Parallèle.

Abstract

This thesis describes an execution environment for parallel machines without shared memory. A parallel programming model based on message passing, with a special shared memory. In this model, process communication occurs indirectly, via ports, and the processes use barriers for synchronization. All the entities of the system, such as processes, ports and barriers, are created dynamically and loaded on any processor of the network of processors.

The implementation architecture of our model is a systematic realization of the client/server model. An implementation is proposed in a Supernode parallel machine as a parallel micro kernel. The principal parallel micro kernel component is a minimal remote procedure call mechanism.

Key words : Distributed memory parallel machine, Parallel programming model, Execution environment, Client/server model, Remote procedure call, Parallel Micro Kernel.