Les entrées-sorties dans les architectures massivement parallèles

Harold Castro

THESE

présentée par

Harold CASTRO

pour obtenir le grade de

DOCTEUR
de l'INSTITUT NATIONAL POLYTECHNIQUE de GRENOBLE

(Arrêté ministériel du 30 Mars 1992)

Spécialité : Informatique

Les Entrées/Sorties dans les architectures massivement parallèles

Date de soutenance : 23 novembre 1995

Composition du jury :

Président :	Guy MAZARE
Rapporteurs :	Claude BOKSENBAUM
	André SCHIPER
Examinateur :	Traian MUNTEAN

Thèse préparée au sein du Laboratoire de Génie Informatique

Remerciements

C’est avec plaisir que je me donne à la tâche de remercier ceux qui ont fait possible que cette thèse voit le jour :

Guy Mazaré, directeur de l’ENSIMAG, pour m’avoir fait l’honneur de présider mon jury.

Claude Boksenbaum, professeur à l’université de Montpellier, et André Schiper, professeur à l’Ecole Polytechnique Fédérale de Lausanne, pour avoir accepté de rapporter ce travail.

Traian Muntean, mon directeur de thèse, pour avoir eu confiance en moi en m’acceptant dans son équipe. Je le remercie pour avoir su m’initier dans ce vaste monde de la recherche.

Grand merci à Robert Despons et Léon Mugwaneza, qui sont d’abord mes amis, ensuite mes colocataires, et finalement mes collègues de travail, pour avoir lu et relu mon manuscrit et pour avoir fait de lui une thèse d’informatique écrite en français.

Tous les autres membres, passés et présents de l’équipe SYMPA, pour leur aide constante et leur SYMPAthie, qui ont facilité que ce travail arrive à bon terme.

Mes chers copains de Grenoble et environs qui ont fait que ces 5 années passées en France, restent comme un des plus beaux souvenirs dans ma mémoire.

L’ES2I, Ecole Supérieur d’Ingénieurs en Informatique, son staff d’enseignants et plusieurs de ses élèves, qui m’ont particulièrement bien accueilli et qui m’ont permis de surmonter cette difficile dernière année de rédaction à Marseille.

Patricia et mes copains en Colombie pour m’avoir gardé une place dans leur coeur malgré la distance.

Et tout naturellement ma famille, qui m’a accompagné et encouragé pendant toutes ces années d’études.

Résumé

Il est reconnu aujourd'hui que pour un grand nombre d'applications les performances globales des systèmes sont fortement limitées faute d'un transfert suffisamment rapide entre les unités de calcul et les dispositifs de stockage. L'idée développée au long de cette thèse est qu'il est possible de réaliser un système d'E/S universel et performant dans un environnement extensible si l'on respecte quelques principes dans sa conception. Pour ce faire, il est nécessaire d'y faire participer le matériel, le système d'exploitation, le système de fichiers et les utilisateurs, chacun au niveau approprié. Dans l'architecture logicielle proposée, nous montrons comment une bonne coopération entre le système de fichiers, le système d'exploitation et les utilisateurs permet l'existence d'une couche logicielle efficace pour les E/S qui garantisse un équilibre entre les services généraux et les fonctionnalités spécifiques.

Notre travail intègre toutes les composantes d'un sous-système d'E/S. En premier lieu, nous choisissons une architecture matérielle extensible adéquate aux divers types de demandes d'E/S observés dans les applications parallèles. Nous présentons une architecture universelle et extensible qui permet de maximiser l'exploitation du parallélisme.

En deuxième lieu, nous utilisons ParX, un micro-noyau parallèle développé à l'intérieur de notre équipe, pour fournir les mécanismes de base à l'exécution d'un système de fichiers parallèle comme serveur global du système. Nous concrétisons d'abord certaines extensions indispensables pour mieux adapter ParX aux besoins des E/S parallèles, et ensuite, afin d'exploiter la projection des fichiers dans l'espace d'adressage, nous développons des mécanismes originaux, nécessaires à l'implémentation d'un espace d'adressage commun dans une architecture extensible à mémoire distribuée.

En troisième lieu, nous introduisons les principes de base qui doivent être respectés afin de concilier la généralité et les hautes performances dans la conception d'un système de fichiers parallèle extensible. L'architecture du système de fichiers proposée à la fin du rapport est le résultat de l'application de ces principes.

Mots clés : Entrées/Sorties parallèles, système de fichiers parallèles, système d'exploitation parallèles, architectures masivemet parallèles, architectures micro-noyau, extensiblité.

Abstract

It is now generally accepted that for a large number of applications a system's global performance is affected if the transfer rate between computing units and storage devices is not fast enough. Throughout this thesis, we develop the idea that it is possible to design a universal and efficient I/O system within a scalable context if some given directives are followed from its conception. Our research integrates all of an I/O system's components. We first choose a hardware architecture which is suitable for the diverse types of I/O requests found in parallel applications. We present an architecture that is both universal and scalable that allows maximising parallelism. Secondly, we use our own micro-kernel, ParX, that offers basic mechanisms for a parallel file system's execution. For this, we first propose some extensions which are indispensable for better adapting ParX to the requirements of parallel I/O's, and later, to exploit file mapping onto address spaces, we develop original mechanisms which are necessary for implementing a common address space in a scalable distributed memory architecture. Finally, we introduce the basic principles that must be observed to reconcile generality and high performances in the design of a parallel and scalable file system. The structure of the file system proposed at the end of this document is the result of the application of these principles.

Key words: parallel Input/Output, parallel file systems, parallel operating systems, massively parallel architectures, micro-kernel architectures, scalability.

Sommaire

Chapitre 3

Chapitre 4

Les Systèmes Gestionnaires de Fichiers 51

Chapitre 5

Le système d'exploitation 77

Chapitre 7

Conception d'un système de fichiers massivement parallèle 135

Chapitre 8

Chapitre 1
Introduction

La puissance de calcul des ordinateurs modernes permet à présent de résoudre des problèmes qui manipulent une énorme quantité de données, et qui nécessitent de hautes vitesses de transfert entre les disques et la mémoire vive. Les opérations d'Entrée/Sortie (E/S) sur disque ont toujours été plus lentes que les opérations de calcul, et malgré les progrès des unités disque les toutes dernières années, il est admis que cette différence de vitesse ne peut que s'agrandir à l'avenir. Ce problème est connu comme la crise des E/S (*I/O crisis*) [PGK88].

L'évolution de la puissance de calcul des ordinateurs n'a pas été toujours suivie par des transformations importantes au niveau des sous-systèmes d'E/S. En effet, dans le souci de pouvoir exécuter les applications toujours plus vite, les concepteurs des machines informatiques ont mis essentiellement l'accent sur la vitesse de calcul des processeurs. Résultat de cette course à la puissance, le parallélisme représente aujourd'hui non seulement la technologie qui permet d'obtenir les puissances de calcul les plus élevées (on annonce déjà des architectures dont la vitesse de calcul dépasserait les teraflops[1]) ; mais, également important, un rapport coût/performances qui est nettement supérieur à celui des machines séquentielles.

Le postulat de Gene Amdhal [Am67] est devenu ainsi peu à peu un mythe de l'informatique : un ordinateur nécessite 1 Mégabit par seconde de puissance d'E/S pour chaque MIPS (Million d'Instructions Par Seconde) de puissance de calcul. Les applications dont les performances sont pénalisées à cause du non-respect de ce postulat se font de plus en plus nombreuses ; des applications telles que la modélisation des flux, la modélisation moléculaire, le traitement de données sismiques et la simulation tactique en sont quelques exemples [In89].

Face à cette évidence, la recherche dans le domaine des E/S s'est accélérée ces dernières années et, comme pour la puissance de calcul, c'est encore le parallélisme qui propose la solution la plus prometteuse. Le principe des E/S parallèles est basé sur la diversification des voies d'E/S. Chaque opération d'E/S est divisée en plusieurs processus d'E/S, où chaque processus suit une

[1] 10^{12} opérations en calcul flottant par seconde

voie différente pour le transfert des données entre les disques et la mémoire vive. L'application de cette technique dans les architectures parallèles permet d'accéder aux données sur disque en parallèle par des voies d'accès indépendantes.

Les progrès les plus remarquables dans la recherche sur les E/S parallèles portent sur l'aspect matériel des sous-systèmes d'E/S et en particulier sur l'organisation de disques parallèles. Il existe aujourd'hui plusieurs niveaux de répartition des données d'un fichier sur un ensemble de disques, chacun avec son propre degré de fiabilité ; dans [CLGK94], P. Chen et al. présentent un historique de ce développement. D'autres domaines, moins explorés, dans la partie matériel sont le placement de caches disque et la structure du réseau d'interconnexion qui relie les dispositifs de stockage aux unités de calcul.

Cependant, tout comme les machines parallèles ne peuvent pas garantir à elles seules de hautes performances de calcul, le matériel des sous-systèmes d'E/S ne peut pas apporter une réponse complète à la crise des E/S. Une solution "intégrale" est nécessaire, et l'aspect logiciel commence à être étudié de façon détaillée. Les concepts du parallélisme s'étendent maintenant au système de fichiers, et de plus en plus, des projets de recherche proposent des systèmes de fichiers parallèles qui visent à exploiter les capacités des sous-sytèmes d'E/S modernes.

1.1 Contribution de cette thèse

Cette thèse fait partie des axes de recherche de l'équipe SYMPA (Systèmes Massivement PArallèles) qui s'intéresse tout particulièrement au support système à l'exécution de programmes dans les architectures massivement parallèles. ParX [CEMW93, La91, Mu+89] est un micro-noyau parallèle autonome et universel, conçu et développé par notre équipe dans le but d'atteindre des hautes performances pour l'exécution d'applications parallèles, à partir d'une construction générique de mécanismes pour la gestion du parallélisme, de la communication et des ressources dans les machines parallèles hétérogènes et extensibles (*scalability*). Les concepts d'universalité, d'extensibilité et des hautes performances dans la conception des systèmes parallèles sont donc les lignes directrices de notre travail.

Les caractéristiques exigées aux sous-systèmes d'E/S sont très diverses en fonction de l'application. Il existe des applications qui exécutent un grand nombre d'opérations d'E/S sur des données de petite taille (p. ex. les bases de données) et des applications avec moins d'opérations d'E/S, mais sur des données de grande taille (p. ex. le traitement d'images). En outre, des applications avec des contraintes temporaires, multimédia et autres, posent de nouveaux défis aux concepteurs des systèmes d'E/S.

Contrairement à la plupart des projets de recherche sur les E/S, où la tendance actuelle est de cibler un ensemble précis d'applications, dont le comportement en matière d'E/S est fortement ressemblant, nous nous intéressons aux systèmes d'E/S universels, c'est-à-dire, des systèmes qui ne favorisent pas un type d'application aux dépens d'un autre. Dans notre travail nous visons alors un ensemble très large d'applications parallèles, et ne faisons aucune hypothèse sur les types de requêtes qu'elles génèrent.

L'idée développée au long de cette thèse est qu'il est possible de réaliser un système d'E/S universel et performant dans un environnement extensible si l'on respecte quelques principes dans sa conception. Pour ce faire, il est nécessaire d'y faire participer le matériel, le système d'exploitation, le système de fichiers et les utilisateurs, chacun au niveau approprié. Dans l'architecture logicielle proposée, nous montrons comment une bonne coopération entre le système de fichiers, le système d'exploitation et les utilisateurs permet l'existence d'une couche logicielle efficace pour les E/S qui garantisse un équilibre entre les services généraux et les fonctionnalités spécifiques.

Notre travail intègre toutes les composantes d'un sous-système d'E/S. En premier lieu, nous choisissons une architecture matérielle extensible adéquate aux divers types de demandes d'E/S observés dans les applications parallèles. Pour cela nous proposons une première classification des architectures parallèles en fonction de leur sous-système d'E/S. A partir de cette classification nous identifions les avantages et les inconvénients de chaque type d'architecture par rapport aux motivations de notre recherche. Nous dégageons ainsi une architecture universelle et extensible qui permet de maximiser l'exploitation du parallélisme.

En deuxième lieu, nous utilisons ParX comme micro-noyau pour fournir les mécanismes de base à l'exécution d'un système de fichiers parallèle comme serveur global du système. Nous concrétisons d'abord certaines extensions indispensables pour mieux adapter ParX aux besoins des E/S parallèles, et ensuite, afin d'exploiter la projection des fichiers dans l'espace d'adressage, nous développons des mécanismes originaux, nécessaires à l'implémentation d'un espace d'adressage commun dans une architecture extensible à mémoire distribuée.

En troisième lieu, nous introduisons les principes de base qui doivent être respectés afin de concilier la généralité et les hautes performances dans la conception d'un système de fichiers parallèle extensible. L'architecture du système de fichiers proposée à la fin du rapport est le résultat de l'application de ces principes.

1.2 Organisation de la suite du rapport

La première partie de ce rapport est consacrée à l'aspect matériel des E/S, où l'on étudie au chapitre 1 les architectures des sous-systèmes d'E/S et le principe des E/S parallèles ; et au chapitre 2 les unités de stockage, plus particulièrement les vecteurs de disques qui sont au cœur des solutions matérielles développées pour résoudre la crise des E/S.

Le chapitre 4 présente l'état de l'art des systèmes de fichiers parallèles qui constituent la composante logicielle principale des sous-systèmes d'E/S. Ce chapitre commence par une courte description des systèmes de fichiers répartis afin de souligner les différences de fond entre ceux-ci et les systèmes de fichiers proposés pour les environnements parallèles.

Ensuite, le chapitre 5 introduit le micro-noyau ParX, les extensions proposées et son approche à base de machines virtuelles pour offrir à chaque application l'abstraction la plus adaptée à son exécution. Le chapitre 6 développe la machine virtuelle à mémoire virtuellement partagée et les mécanismes nécessaires à son implémentation. La projection des fichiers sur cet espace d'adressage commun permet de réutiliser les mécanismes développés pour la localisation et la gestion de la cohérence de leurs données.

Finalement le chapitre 7 se consacre à la définition des directives pour la conception d'un système de fichiers parallèle et à la présentation du système de fichiers construit à partir de ces directives et son intégration au système d'exploitation parallèle développé autour de ParX.

Chapitre 2
Architecture matérielle des systèmes d'Entrées/Sorties

Les dernières décennies ont été marquées par une course à la puissance de calcul dans les ordinateurs qui contraste nettement avec les faibles améliorations apportées aux systèmes d'E/S. En effet, l'influence des E/S pour certaines applications a été négligée et les principales innovations ont été portées au niveau des algorithmes des applications, des compilateurs et surtout des vitesses de calcul des processeurs. Malgré les efforts consacrés aux sous-systèmes d'E/S ces toutes dernières années, nous observons aujourd'hui un déséquilibre pénalisant entre les capacités des processeurs et celles des systèmes d'E/S.

L'importance d'un équilibre entre la puissance de calcul et le débit des E/S est largement reconnue depuis longtemps [Am67]. En effet, pour un grand nombre d'applications les performances globales des systèmes sont fortement limitées faute d'un transfert suffisamment rapide entre les unités de calcul et les dispositifs de stockage.

Les architectures parallèles sont particulièrement sensibles à ce déséquilibre ; leur puissance de calcul étant très importante, elles nécessitent un sous-système d'E/S très performant pour qu'une application puisse tirer tout le profit possible de ces systèmes. Augmenter la taille de la mémoire vive des processeurs n'est plus suffisant pour dissimuler leur handicap actuel, et de nouvelles organisations, mieux structurées et mieux adaptées, doivent donc être proposées pour permettre aux sous-systèmes d'E/S de répondre efficacement aux évolutions des puissances de calcul.

Par ailleurs, l'architecture des sous-systèmes d'E/S doit être capable d'augmenter l'efficacité des opérations d'E/S en fonction de la croissance des vitesses de calcul. En effet, il est nécessaire que le sous-système d'E/S accélère la vitesse d'accès aux informations stockées sur disque dans les mêmes proportions que les capacités de calcul sont accrues. De ce fait, le concept

d'extensibilité s'applique non seulement à la puissance de calcul de la machine, mais également à son sous-système d'E/S.

Dans ce chapitre nous présentons un état de l'art sur les organisations matérielles des sous-systèmes d'E/S. Nous cherchons plus particulièrement à dégager l'approche qui équilibre le mieux le ratio puissance de calcul/débit des E/S. Cette étude a comme cadre les machines massivement parallèles, et dans un souci de clarté nous commençons par détailler les différentes approches architecturales pour le parallélisme massif et ensuite nous y grefferons les concepts d'E/S.

2.1 Les Architectures massivement parallèles

Le paradigme du parallélisme massif a été couramment défini de la façon suivante :

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

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

Cette définition a fait concentrer la plupart des efforts conceptuels sur la capacité d'adaptation de la puissance de calcul d'une architecture, négligeant ainsi celle des sous-systèmes d'E/S. Aujourd'hui l'impact des E/S sur les performances globales des systèmes informatiques est amplement reconnu, et donc l'adaptation de la puissance de calcul doit être accompagnée de celle du sous-système d'E/S. Nous considérons alors qu'un système qui se veut extensible, doit facilement adapter ses capacités de calcul et d'E/S aux besoins des applications dans les deux domaines.

Dans cette section nous passons en revue les tendances actuelles de la recherche sur les architectures massivement parallèles auxquelles s'applique notre travail. Nous introduisons trois groupes majeurs d'architectures : celles orientées parallélisme des données, celles orientées parallélisme des traitements, et une combinaison hybride des deux (parallélisme de contrôle). La présentation des réseaux d'interconnexion est volontairement omise mais un aperçu général,

ainsi qu'une description complète des machines pour une classe ou pour une autre peuvent être trouvés dans le livre "*Highly Parallel Computing"*, de G.S. Almasi et A. Gottlieb [AG94][1].

2.1.1 Architectures orientées parallélisme des données

Les architectures orientées parallélisme des données, couramment appelées SIMD (**S**ingle **I**nstruction stream - **M**ultiple **D**ata stream) ou plus récemment SPMD (**S**ingle **P**rocess - **M**ultiple **D**ata), forment le groupe avec l'histoire la plus longue. Ce modèle fut en premier lieu utilisé sur des données (parallélisme de données) représentées par des vecteurs, structure très caractéristique de beaucoup de problèmes scientifiques, et il trouvait sa réalisation dans les machines vectorielles. Cette technique de calcul, encore très utilisée aujourd'hui, consiste à décomposer les données sous forme de vecteurs et à appliquer de manière synchrone à toutes les données d'un vecteur le même traitement.

Depuis, ce modèle a vu élargir son champ d'application à d'autres domaines et le modèle SIMD est maintenant utilisé pour de nombreuses architectures de machines. Le principe général est que de nombreux processeurs exécutent simultanément la même instruction ; l'ensemble des processeurs est ainsi fortement synchronisé et de cette manière tous ceux qui sont actifs exécutent la même action au même instant sur leur donnée respective. Ce synchronisme est rendu possible grâce à un mécanisme de contrôle global : le séquenceur.

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

L'architecture est organisée autour d'un réseau d'interconnexion qui permet la communication entre nœuds par l'intermédiaire de liaisons point à point. Aussi bien la structure du réseau, typiquement régulière, que les mécanismes de transfert des données, qui se réduisent normalement à l'échange entre processeurs directement connectés (voisins), sont essentiellement déterminés par les besoins qui découlent du type de parallélisme exploité par cette architecture. C'est le parallélisme de données, dans lequel l'ensemble des données à traiter est découpé en données élémentaires chacune traitée par un processeur. Cette décomposition met généralement

[1] Dans la suite, la référence [AG94] est sous-entendue comme référence de base pour la description des machines dont la bibliographie n'est pas mentionnée.

en évidence des propriétés de localité des données par rapport aux traitements qu'elles sont susceptibles de subir, par conséquent, un processeur donné n'a besoin d'interagir qu'avec un nombre réduit d'autres processeurs qui sont, le plus souvent, ses voisins physiques dans le réseau.

Les Connection Machines [CM88] CM-1 et CM-2 sont les exemples les plus connus de cette architecture. Ici un grand nombre de processeurs très simples (jusqu'à 65536 processeurs d'un bit) est utilisé pour implanter des solutions massivement parallèles. Avec un nombre de processeurs beaucoup moins important, L'ILLIAC IV conçue dans les années 60 à l'université d'Illinois, la GF11 d'IBM et les machines vectorielles telles que les séries Cray sont également représentatives du modèle SIMD.

2.1.2 Architectures orientées parallélisme des traitements

Appelées couramment architectures MIMD (**M**ultiple **I**nstruction stream - **M**ultiple **D**ata stream), les architectures orientées parallélisme des traitements sont aujourd'hui à la base des recherches les plus actives en conception de machines parallèles. Ici chaque processeur a accès à une mémoire vive (RAM) importante, ce qui lui permet d'avoir son propre programme et ainsi d'exécuter ses instructions indépendamment des autres. C'est l'asynchronisme qui caractérise le fonctionnement de ces architectures.

Les processeurs sont plus complexes et plus puissants que ceux utilisés dans les architectures SIMD. Chaque processeur peut intégrer une unité arithmétique et logique, une unité de calcul flottant, une anté-mémoire (mémoire cache) ou encore une ou plusieurs unités dédiées aux communications. De par le caractère indépendant de ses unités, une machine MIMD a besoin d'un système de communication plus flexible que celui des architectures SIMD. La façon de communiquer entre processeurs détermine donc ces architectures et deux groupes importants peuvent être distingués : les architectures sans mémoire commune pour lesquelles chaque processeur n'a accès qu'à une mémoire locale privée (figure 2.1a) et les architectures à mémoire commune où tous les processeurs partagent plusieurs bancs de mémoire (figure 2.1b).

2.1.2.1 Architectures MIMD à mémoire commune

L'essence du modèle à mémoire commune ou partagée est que tous les processeurs ont un accès équitable à un grand et unique espace d'adressage[1]. Physiquement, la mémoire est formée d'un

[1] Une sous-classe de ces architectures sont les architectures NUMA (**N**on **U**niform **A**ccess **M**emory) où la caractéristique d'accès équitable n'est plus respectée. Le modèle NUMA offre l'espace unique d'adressage à

certain nombre de modules séparés. Un réseau d'interconnexion permet l'accès de chaque processeur à chaque banc mémoire ; un tel réseau correspond généralement à un graphe bipartite ayant d'un coté l'ensemble des processeurs et l'ensemble des modules mémoires de l'autre.

P = Processeur
M = Mémoire

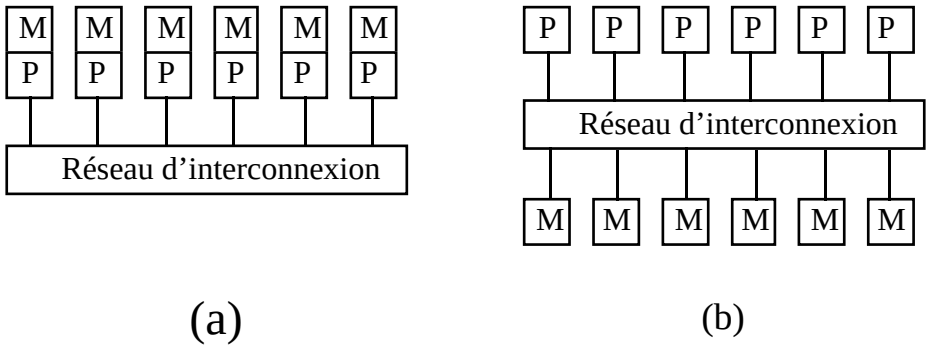


Figure 2.1 Architectures parallèles avec et sans mémoire commune.

L'espace d'adressage partagé laisse le choix du modèle de programmation aux utilisateurs et facilite ainsi l'utilisation de ces architectures : le modèle d'échange de messages peut être simulé avec des tampons de communication dans la mémoire commune, et le paradigme des variables partagées est directement utilisable ; il suffit pour cela de faire appel aux techniques classiques des compilateurs pour restreindre la visibilité d'une variable non partageable.

Comme toutes les interactions inter-processus (communications et accès aux variables partagées) se font à travers la mémoire, et même si des optimisations peuvent être apportées, la mémoire commune devient rapidement un goulot d'étranglement du système ; de plus, il faut résoudre le problème de cohérence de l'information qui s'y trouve (conflits d'accès simultanés). Malgré l'introduction de caches pour réduire le trafic vers ou depuis la mémoire, mais qui entraînent de nouveaux problèmes pour la cohérence globale de ces caches, la capacité d'extension de ces architectures reste faible. De plus des contraintes technologiques des bus de communication limitent fortement le nombre de processeurs qu'il est possible de connecter. En effet, la bande passante des bus de communication actuels ainsi que le nombre d'unités qu'on peut y attacher ne permet pas d'envisager la conception d'une machine au-delà de la centaine de processeurs.

Afin d'augmenter leur extensibilité, ces architectures sont souvent divisées en partitions, et c'est au niveau des partitions qui se trouve la mémoire physiquement commune. Des mécanismes matériels sont mis en œuvre pour garantir l'accès aux données à l'extérieur d'une partition. Comme exemples de ces architectures on trouve la machine DASH conçue à l'université de

partir d'un ensemble de mémoires privées, mais pour chaque processeur l'accès à une mémoire distante est plus coûteux que l'accès à sa propre mémoire privée.

Stanford [LLGW92], Alewife du MIT, la BBN Butterfly, et l'IBM RP3. Les machines KSR de Kendall Square [KSR92], FLASH de l'université de Stanford [KOHH94] et le Cray T3D sont aussi admises dans cette classe bien que la base matérielle de ces machines ne comporte pas une véritable mémoire commune aux processeurs. Dans ce dernier groupe de machines, la mémoire physique est composée d'un ensemble de mémoires privées, et un mécanisme matériel est intégré afin d'offrir aux utilisateurs la vision d'une mémoire commune partagée par tous les processeurs.

2.1.2.2 Architectures MIMD sans mémoire commune

Au même titre que les architectures à mémoire commune sont les préférées des programmeurs, les architectures à mémoire distribuée sont le choix premier d'un concepteur de machines parallèles. En effet la construction d'une machine massivement parallèle à mémoire privée est plus simple car le problème des accès simultanés à une mémoire commune ne se pose plus, et malgré les problèmes inhérents au contrôle correct des échanges de messages entre processeurs [Go91, Mu93], la taille de la machine peut être aisément modulée.

Dans ce type d'architecture, chaque processeur dispose d'un espace d'adressage qui lui est propre ; un processeur ne peut accéder à la mémoire locale d'un autre que par échange de messages à travers un réseau de communication. L'accès à une donnée en dehors de sa mémoire privée est donc largement plus coûteux et les caractéristiques du réseau d'interconnexion des processeurs prennent une importance capitale. Ces architectures sont en conséquence souvent comparées par rapport au réseau qui relie les unités de calcul entre elles.

Le réseau d'interconnexion définit la structure spécifique d'un ordinateur MIMD ; c'est lui qui caractérise les facilités offertes pour la communication entre les processeurs. Sa participation dans l'activité globale de la machine est si fondamentale qu'il a une influence cruciale sur ses performances. Pour les architectures massivement parallèles, la minimisation de la distance par une maximisation de la connectivité permanente se heurte à l'impossibilité de réalisation pratique, des compromis sont donc nécessaires. Deux approches se dégagent : l'une consiste à connecter les processeurs à travers un réseau statique d'une structure régulière, réseau figé lors de la construction de la machine, l'autre se base sur des réseaux dits réconfigurables, où le matériel essaie de s'adapter le mieux possible aux besoins de communication de l'application.

Différents types de topologies pour les réseaux d'interconnexion statiques ont été amplement étudiés : l'hypercube (Caltech Cosmic Cube, Intel iPSC et iPSC/860, NCube/ten et NCUBE 2), la grille (Intel Paragon XP/S, PAX-128, IBM Victor, IBM WRM), les arbres (Teradata Database

Computer, CM-5 [Le+92]), les réseaux basés sur un bus global (Tandem) et les réseaux multi-étages (IBM 9076 SP1).

Dans les réseaux réconfigurables on ne peut plus parler d'une topologie particulière : toute topologie pourrait éventuellement être réalisée par un réseau réconfigurable, dans la limite du degré de connectivité des processeurs. Les réseaux réconfigurables les plus connus sont le crossbar (Supernode [Wa91]), les réseaux multi-étage programmables et le réseau Omega [AG94].

2.1.3 Architectures parallèles hybrides

Certaines machines parallèles ne rentrent pas exactement dans l'une ou l'autre des deux catégories précédentes. Deux nouveaux groupes peuvent être identifiés bien qu'ils soient encore des projets de recherche. Le premier est constitué par des machines à très longues instructions (VLIW pour **V**ery **L**ong **I**nstruction **W**ord) et le deuxième est représenté par les machines SIMD multiples (MSIMD).

Un multiprocesseur à très longues instructions a une architecture similaire au SIMD, dans le sens où tous les processeurs sont synchronisés par un séquenceur central. Cependant, les instructions que les processeurs exécutent au même instant ne sont pas forcément les mêmes. L'idée de ces architectures est de rapprocher le matériel d'un certain type de compilateurs et d'optimiser ainsi l'utilisation de chaque processeur d'une architecture parallèle. Pour ce faire, le compilateur regroupe des instructions simples qui ne peuvent pas s'exécuter en parallèle, et marque ce groupe d'instructions comme étant une instruction longue. Ensuite, le séquenceur de l'architecture VLIW distribue ces groupes d'instructions aux différents processeurs de la machine pour leur exécution. Les machines Multiflow TRACE, CHoPP et IBM YSE/EVE en sont des exemples de réalisation.

Le modèle MSIMD peut être vu comme un ensemble de machines SIMD regroupées sous une seule architecture. L'idée est d'offrir, grâce à un réseau réconfigurable ou à une topologie arborescente du réseau, la machine SIMD la mieux adaptée à chaque application. On peut ainsi avoir une machine divisée en sous-configurations multiples, où chacune s'exploite en mode SIMD. Dans les machines aux réseaux arborescents nous retrouvons l'UNC Cellular Computer, et la Columbia NON-VON. L'UT TRAC et la machine PASM de l'université de Purdue utilisent un réseau réconfigurable pour déterminer les partitions.

La figure 2.2 résume les différences, du point de vue flot de contrôle, entre les architectures SIMD, MIMD, VLIW et MSIMD ; ici les identificateurs A, B, C et D représentent des

instructions processeur. On peut observer que trois de ces architectures (SIMD, VLIW et MSIMD) sont régies par un séquenceur global, mais que les possibles instructions à exécuter varient d'une architecture à l'autre.

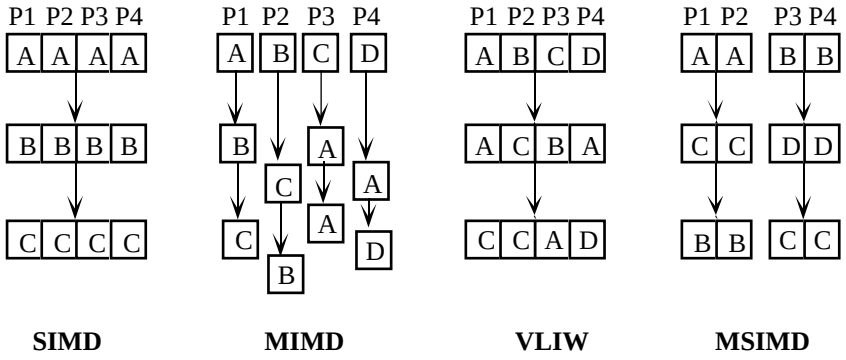


Figure 2.2 Flot de contrôle des diverses architectures.

De cette courte présentation des architectures parallèles, il ressort que les architectures MIMD sont les architectures les plus générales et les moins restrictives pour leur exploitation ; et parmi celles-ci, les architectures à mémoire privée sont celles qui répondent le plus naturellement aux exigences d'extensibilité du parallélisme massif. Les caractéristiques d'extensibilité, universalité et des hautes performances que nous établissions comme lignes directrices de notre travail lors de l'introduction de ce rapport nous imposent donc de prendre tout particulièrement en compte ces architectures. Dans la suite nous considérons alors ce type d'architecture comme la base matérielle de notre travail.

2.2 Les Entrées/Sorties parallèles

Dans les systèmes traditionnels les opérations d'E/S sont réalisées de façon purement séquentielle et synchrone. Un processus utilisateur exécute un appel système pour effectuer l'E/S, le système d'exploitation transmet la requête au sous-système d'E/S, et ce dernier active les dispositifs d'E/S et effectue l'opération proprement dite. Le processus qui est à l'origine de la requête reste suspendu en attendant la fin de l'opération. La figure 2.3 montre un système classique où les données générées par une opération d'E/S sur une unité de disque suivent une voie unique pour se déplacer, dans les deux sens, entre le disque et la mémoire vive. Ici les fichiers sont stockés par blocs, et tous les blocs d'un fichier se trouvent sur le même disque ; la lecture/écriture d'un fichier est réalisée aussi par blocs, et les tampons du système servent d'anté-mémoires pour loger ces blocs et ainsi limiter les accès au disque. Dans les grosses machines (*main frames*), cette voie est typiquement constituée d'un processeur d'E/S, un canal

d'E/S et un contrôleur de disque. Les performances de ces systèmes sont dominées par la vitesse des opérations de lecture/écriture sur le disque.

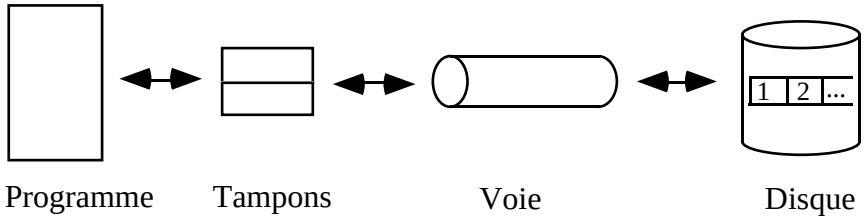


Figure 2.3 E/S séquentielles standards.

Plusieurs optimisations peuvent être apportées à cette architecture simple. Le modèle client/serveur par exemple permet de traiter diverses requêtes d'E/S en parallèle, chacune en provenance d'un client différent. Les E/S continuent à être synchrones du point de vue du client mais les sous-systèmes d'E/S peuvent mettre à jour un tampon tout en exécutant une autre opération sur le disque. Cette solution impose néanmoins l'exécution une à une de toutes les opérations d'E/S, même si elles sont générées en parallèle.

De la même façon qu'une application est décomposée en plusieurs tâches pour exploiter son parallélisme, une opération d'E/S peut être divisée en plusieurs processus d'E/S pour être exécutés en parallèle. Le principe est similaire à celui des pipelines utilisés dans les unités arithmétiques et logiques des super-calculateurs. La figure 2.4 présente schématiquement le principe des E/S parallèles. Le modèle est basé sur la diversification des voies d'E/S. C'est ainsi que l'on obtient plusieurs unités de disque, chacune avec une voie d'accès différente. Une opération d'E/S sera divisée donc en plusieurs processus d'E/S, le nombre de processus à générer sera une fonction du nombre de voies disponibles, et chaque processus suivra une voie différente pour transférer les données entre les disques et la mémoire vive.

Pour que le découpage d'une opération d'E/S soit efficace, il faut que les tâches associées aux processus d'E/S soient indépendantes les unes des autres. Cette indépendance est indispensable puisque les données accédées par chaque processus doivent emprunter des voies distinctes et ceci pendant tout le transfert.

Le but d'une opération d'E/S sur une unité de stockage est le transfert d'une zone de données entre la mémoire vive et le(s) périphérique(s) de stockage (normalement les disques). La trajectoire empruntée par cette zone commence (ou termine) dans les unités de disques, et si l'on veut garder l'indépendance d'une voie pendant tout le parcours des données, il faut que la zone soit également éclatée selon les différentes unités de disque (cf. figure 2.4).

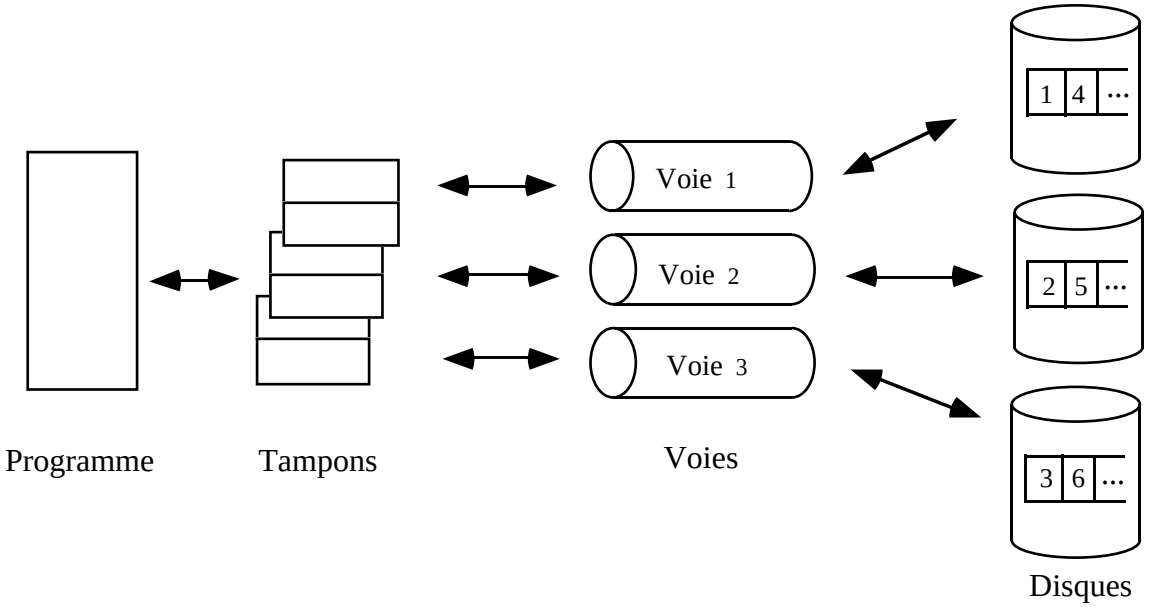


Figure 2.4 E/S avec trois voies parallèles.

Dans un système de stockage classique un fichier n'est pas éclaté sur plusieurs disques, à moins que sa taille ne l'exige. Lorsqu'il s'agit d'un système d'E/S parallèle, les blocs d'un même fichier sont stockés sur les différents disques afin d'être accédés en parallèle ; c'est l'*entrelacement des fichiers.* Le nombre d'unités de disques utilisées détermine le degré de parallélisme physique du système, c'est-à-dire, le nombre de voies différentes que les données pourront emprunter lors d'un transfert. Nous reviendrons sur l'entrelacement de fichiers, lors de la présentation de l'organisation des données au chapitre 3.

Dans un sous-système d'E/S de degré 3 par exemple, le premier bloc d'un fichier pourrait être enregistré sur le premier disque, le deuxième bloc sur le deuxième disque, le troisième bloc sur le troisième disque, le quatrième bloc sur le premier disque et ainsi de suite. Une opération de lecture/écriture de 3 blocs consécutifs ou plus du fichier empruntera donc les trois voies disponibles en parallèle. De plus, le parallélisme offert par les différentes voies d'E/S permet aussi à plusieurs opérations, lecture ou écriture, si elles sont indépendantes (blocs concernés par les opérations associés à des disques différents) d'être effectuées simultanément. Ainsi, une opération de lecture sur le bloc 2 du fichier peut s'exécuter en parallèle avec une opération d'écriture du bloc 4.

L'unité de découpage peut être choisie en fonction des types d'accès particuliers de chaque fichier. Néanmoins, il existe des configurations matérielles qui peuvent fixer la taille de cette unité. Certaines réalisations existantes offrent aujourd'hui une variété importante d'unités de découpage, cela va de 1 bit à la taille de blocs physiques des disques [PGK88].

La figure 2.5 donne une idée des gains obtenus en vitesse d'accès par l'utilisation d'un sous-système d'E/S parallèle. La procédure standard d'écriture dans un système à quatre tampons est montrée dans la première partie de la figure. La première ligne indique l'utilisation de l'unité de calcul, et le numéro du tampon associé à chaque opération ; ici, le temps mis par l'unité de calcul pour transférer les données entre la mémoire vive et les tampons d'E/S est augmenté par rapport à celui des opérations d'E/S pour clarifier la figure. La deuxième ligne montre l'exécution du programme d'E/S (par le contrôleur et/ou un processeur spécialisé E/S) et l'attente de l'unité centrale d'un signal de terminaison d'E/S. Enfin, la dernière ligne montre le temps de positionnement des têtes du disque (y compris le temps de rotation du disque pour trouver le bon secteur) et le temps d'écriture ou de lecture effective sur l'unité. Avec la procédure standard, on observe qu'au bout de six transferts, tous les quatre tampons sont occupés et qu'une nouvelle opération d'E/S reste bloquée en attendant la libération du tampon 3.

La figure 2.5b montre l'application du même cas avec un sous-système parallèle d'E/S de degré 2. Grâce aux deux voies employées par les données, le temps d'exécution d'une opération est presque réduit de moitié. En effet, les attentes pour des signaux de fin d'E/S par l'unité centrale sont beaucoup moins nombreuses que dans le premier exemple. Les tampons intermédiaires d'E/S sont par conséquent libérés beaucoup plus rapidement ; dans la figure nous observons que même après huit opérations, une nouvelle requête ne serait pas bloquée car deux tampons sont déjà disponibles.

L'évaluation des performances d'un système parallèle d'E/S est néanmoins plus complexe. La charge de chaque processeur, la concurrence des accès, le type des accès, le réseau d'interconnexion entre les disques et les unités de calcul en sont quelque exemples des paramètres qui doivent être pris en compte. Dans [NNSI90], les auteurs font une première évaluation, pour une configuration particulière d'une machine HITAC VOS3/ES1, où ils arrivent à montrer que l'accélération obtenue dans la vitesse des accès est proportionnelle au degré de parallélisme du sous-système d'E/S. Cependant ces résultats ne sont vérifiés que dans des conditions très spécifiques, ce qui ne permet pas de les généraliser.

Par ailleurs les études de M. Livny, S. Khoshafian et H. Boral [LKB87] et d'A.L. Narasimha, Reddy et P. Banerjee [NB89] ont montré que la distribution des fichiers réduit effectivement le temps de séjour d'une requête dans la liste d'attente des opérations d'E/S. Le facteur d'amélioration des performances reste néanmoins dépendant du type d'accès aux données utilisé par les applications. D'autres organisations qui favorisent l'accès en parallèle aux fichiers sont étudiées dans la section 3.3.3.

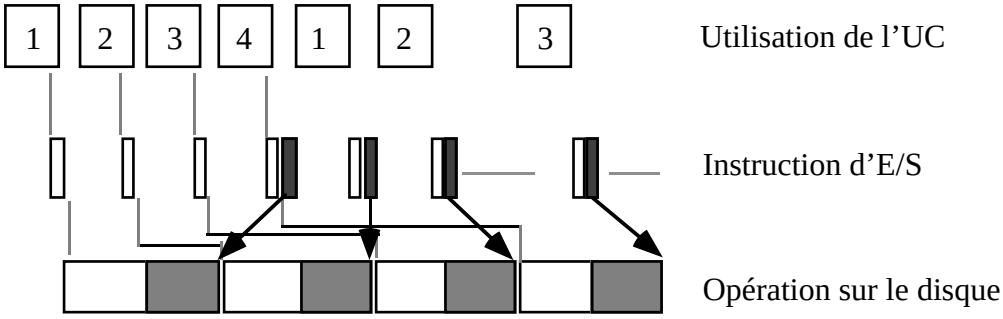


Figure 2.5a Exécution des opérations dans un système d'E/S standard.

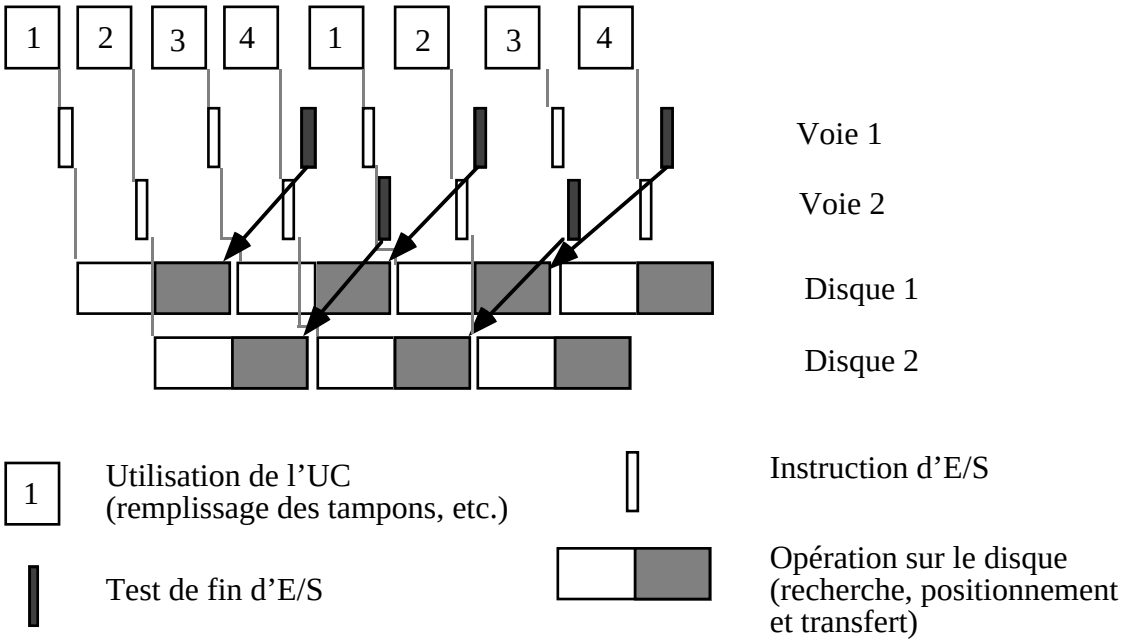


Figure 2.5b Exécution des opérations dans un système d'E/S parallèle de degré 2.

2.3 Architecture des systèmes parallèles d'E/S

L'entrelacement des fichiers s'avère efficace mais le gain effectif en vitesse d'accès reste fortement dépendant de la configuration de tout le système, et pour le cas des systèmes massivement parallèles la contrainte d'extensibilité joue un rôle décisif dans le choix de cette configuration. L'atout du parallélisme massif est qu'il offre une excellente capacité d'extension, surtout en ce qui concerne les nœuds de calcul. Si, en théorie du moins, il est possible d'appliquer le même principe avec les unités de stockage, par adjonction de disques (avec leur voie d'accès indépendante) proportionnellement au nombre de processeurs, dans la pratique des

problèmes comme la distribution des données, la saturation des réseaux ou la tolérance aux pannes contraignent cette possibilité.

La conception d'un système parallèle d'E/S doit également prendre en compte les diverses conditions requises par les applications en matière d'E/S. Augmenter le débit total des E/S n'est parfois pas suffisant et de nouvelles techniques, qui favorisent l'exploitation du parallélisme, sont nécessaires pour réduire le temps de réponse des requêtes concernant des données de petite taille.

L'état naissant des recherches sur ce sujet ne nous permet pas d'étudier une classification reconnue des sous-systèmes d'E/S comme nous l'avons fait pour les architectures parallèles. Nous présentons alors, les tendances architecturales des E/S dans les machines parallèles, et proposons, tel que nous l'avons introduit dans [Ca94], une première classification des architectures d'E/S en suivant la façon dont sont reliés les disques aux processeurs. Nous identifions deux groupes architecturaux : d'un coté les machines à une disposition fixe des unités de stockage où chaque processeur dispose d'une connexion directe à un ou plusieurs disques, et d'un autre coté les configurations plus générales où la communication entre les unités de calcul et les unités de stockage se fait à travers un réseau d'interconnexion.

2.3.1 Architectures à disques locaux privés

Dans [CW93], S. Coleman et R. Watson proposent de doter le sous-système d'E/S des machines parallèles d'un réseau spécialisé pour acheminer les données entre les unités de stockage et les différents blocs de mémoire. Cette extension cherche à éliminer les conflits, nécessairement présents sur un seul réseau, entre le trafic dû à la communication inter-processus des applications et celui propre aux E/S. Dans une telle architecture toute unité de calcul est liée à deux réseaux d'interconnexion, un pour la communication inter-processus et un autre pour transférer les données en provenance, ou en direction, des dispositifs d'E/S. Cette solution est très coûteuse du point de vue architectural et les auteurs n'arrivent pas à déterminer un nombre important d'applications qui pourraient justifier un tel investissement.

Une solution qui va dans le sens de la proposition de Coleman et Watson, est celle qui consiste à relier par une liaison directe chaque processeur à ses propres disques locaux (voir figure 2.6). Ici un mécanisme de DMA (**D**irect **M**emory **A**cces ou dispositif d'accès direct à la mémoire) contrôle, localement au processeur, les transferts entre la mémoire et les unités de disque, et le haut débit du système est assuré par de grands bancs mémoire qui jouent le rôle de tampons d'E/S. Ces architectures ont été conçues avec un souci bien défini : minimiser le temps de

transfert de grands volumes de données entre les disques et les unités de calcul. Aujourd'hui, elles voient leur développement dans les super-calculateurs à dessein bien arrêté.

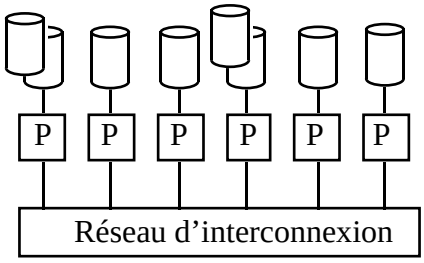


Figure 2.6 Un système d'E/S à connexion locale processeur - disque.

Le regroupement processeur-disque est une conséquence directe du couple processeur-mémoire. En effet, ces architectures ne sont qu'une extension du modèle MIMD à mémoire privée où les disques constituent un niveau de plus dans la hiérarchie des mémoires. Un nœud de la machine devient maintenant un ordinateur complet avec un ou plusieurs processeurs, une mémoire vive et un ou plusieurs disques de stockage.

Pour ces architectures, il est préférable que la plupart des opérations d'E/S émanant d'un processeur soient effectuées le plus souvent possible sur son disque local. Les concepteurs de telles machines s'adressent aux applications pour lesquelles une étude sérieuse du placement des données en disque a été réalisée au préalable, et, plus critique encore, où les données font preuve d'une forte stabilité. C'est le cas par exemple des applications comme la visualisation d'images où chaque partie d'une image est traitée et stockée par un processeur différent.

L'hypothèse de localité n'est cependant remplie que par un ensemble bien réduit d'applications et ainsi son utilisation reste très restreinte. En outre, construire un système où chaque processeur est relié à une unité de stockage limite fortement le nombre de processeurs de la machine (taille physique de la machine, coût de réalisation, etc.). Une conception plus réaliste pour les machines massivement parallèles serait de regrouper, dans un nœud, plusieurs processeurs autour d'une unité de stockage, tel qu'il a été fait dans le sous-système d'E/S de l'hypercube nCUBE [HMSC86].

La machine Many/370 d'IBM est un exemple d'architecture à disques locaux poussée à son extrême [AGHL91]. Many/370 est une machine parallèle construite pour exécuter des applications qui font un usage intense d'E/S. Son prototype est constitué de 8 processeurs, 128 disques de petite taille et un ordinateur hôte. Chaque processeur dispose d'un adaptateur d'E/S qui lui permet de contrôler 16 disques à partir de 4 bus SCSI (**S**mall **C**omputer **S**ystem **I**nterface) différents (cf. figure 2.7). Malheureusement les auteurs ne donnent aucune évaluation globale des performances et se limitent dans leur présentation à comparer les performances des

opérations d'E/S locales et distantes, c'est-à-dire des opérations sur un des disques attachés directement au processeur qui exécute l'E/S et celles qui concernent le disque de l'ordinateur hôte. Il en résulte un débit qui est au moins deux fois plus important pour les transferts locaux que ceux qui nécessitent l'intervention du commutateur.

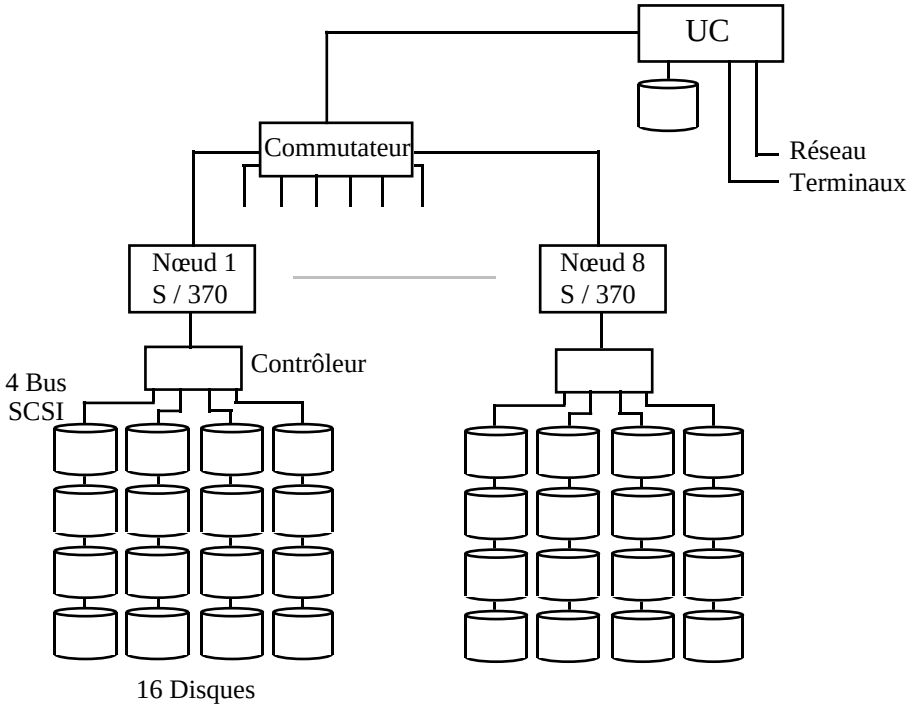


Figure 2.7 Architecture du sous-système d'E/S du IBM Many/370.

2.3.2 Architectures à disques communs

Au même titre que les architectures à disques privés se sont inspirées du modèle MIMD à mémoire privée, les architectures à réseau d'interconnexion pour l'accès aux disques représentent une extension du modèle MIMD à mémoire partagée.

Le principe de base des architectures à disques communs ou partagés est que tout processeur doit avoir un accès équitable à toute information disponible en mémoire secondaire. Pour atteindre cet objectif, elles utilisent des processeurs spécialisés, dits processeurs d'E/S. Ceux-ci sont, à travers le réseau d'interconnexion, les ponts nécessaires pour l'établissement équitable de communications entre unités de stockage et unités de calcul. Un processeur de calcul n'a plus alors d'accès direct à une unité de disque, mais doit adresser une requête à un serveur capable de réaliser toute opération d'E/S.

La figure 2.8 montre la structure d'un système d'E/S parallèle à disques communs. Le réseau d'interconnexion relie deux entités bien définies : les unités de calcul d'un coté et les nœuds d'E/S de l'autre. Un nœud d'E/S est composé d'un processeur spécialisé relié directement au réseau, et d'une ou plusieurs unités de stockage qui sont accessibles aux unités de calcul exclusivement à travers le processeur d'E/S. Les unités de calcul sont supposées dotées de processeurs optimaux (par exemple avec une unité de calcul flottant) pour les applications qui nécessitent une grande puissance de calcul. Les dispositifs de stockage peuvent être très variés et ils sont normalement organisés d'une façon hiérarchique.

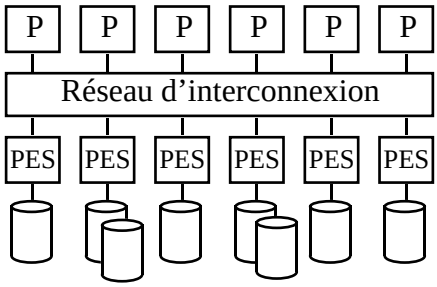


Figure 2.8 Un sous-système d'E/S à disques partagés.

Diverses machines commerciales comme l'hypercube Intel iPSC [AG94], la CM-5 de Thinking Machines [Le+92], la Paragon XP/S de Intel [Me93] et Tera de Teradata Corporation [Al+90] utilisent ce type d'organisation pour leur système d'E/S. La figure 2.9 montre le système d'E/S de la machine CM-5 telle qu'elle est décrite dans [Le+92]. D'une architecture flexible, la machine CM-5 peut être composée d'une dizaine à un millier de processeurs, chacun avec sa propre mémoire. Contrairement aux autres Connection Machines CM-1 et CM-2, la machine CM-5 est une architecture MIMD, bien qu'elle dispose toujours d'un support matériel pour sa programmation en mode SIMD. La machine peut être divisée en plusieurs partitions et dans chaque partition un processeur de contrôle (PC) exécute le système d'exploitation (CMOST) et contrôle l'accès et l'utilisation des unités de calcul (UC). Il existe une partition spéciale dédiée aux E/S avec un PC d'E/S et plusieurs vecteurs de disques (SDA pour **S**calable **D**isk **A**rray ou vecteur de disques extensible). Un SDA est formé d'un ensemble de nœuds de stockage, chacun avec jusqu'à 8 disques de 1.2 Go, 4 contrôleurs SCSI-2, une mémoire tampon de 8 Mo, un processeur et une interface d'accès aux réseaux de données et de contrôle. Le débit d'un SDA est donné par la capacité de l'interface réseau, c'est-à-dire 20 Méga-octets/s, et l'extensibilité du débit global est assuré par la structure de "fat tree" du réseau de données.

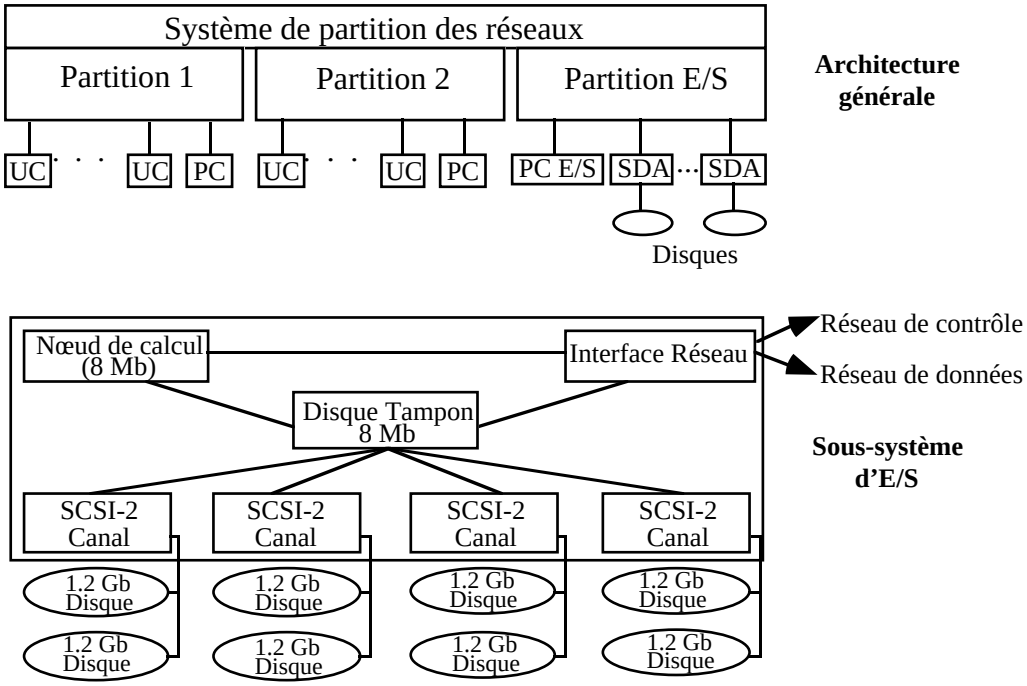


Figure 2.9 Architecture et sous-système d'E/S de la CM-5.

2.3.3 Une architecture universelle pour les E/S

Comme nous l'avons vu lors de l'introduction aux E/S parallèles, quatre éléments composent essentiellement les voies d'accès entre les unités de calcul et les disques, chacun avec une fonctionnalité bien définie (cf. figure 2.10) :

Les unités de calcul (UC) exécutent un ou plusieurs des processus d'une application parallèle. Elles génèrent les requêtes d'E/S.

Les processeurs d'E/S (PES) exécutent les opérations d'E/S. Ils assurent la gestion des fichiers logiques.

Les contrôleurs de disque (CTR) transfèrent les blocs des fichiers entre les PES et les disques.

Les disques sont les dispositifs de stockage de données.

Ces quatre éléments fonctionnels ne sont pas toujours physiquement séparés dans les réalisations actuelles. Seuls les gros ordinateurs disposent d'un composant matériel indépendant pour chaque élément du sous-système. Dans d'autres architectures, des éléments adjacents sont souvent combinés dans un seul composant matériel. Ainsi, les fonctionnalités des UC et des PES sont en général assurées par le même microprocesseur et si l'architecture dispose de PES

séparés des UC, les PES ont souvent un contrôleur intégré. Enfin, il est aussi fréquent de trouver les contrôleurs dans les unités de stockage.

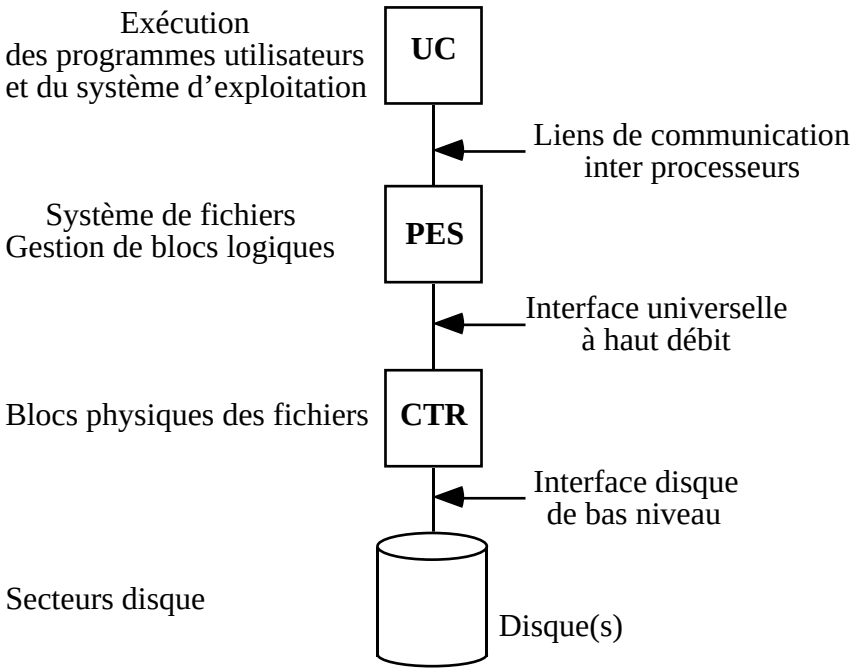


Figure 2.10 Organisation fonctionnelle d'un sous-système d'E/S.

La combinaison des éléments fonctionnels a l'avantage de réduire la longueur de la voie d'accès entre deux éléments non voisins. Des copies de données inter éléments peuvent ainsi être évitées, et les performances d'une opération d'E/S améliorées. Toutefois, la flexibilité du système est moindre car les voies sont fixes et ne peuvent pas être modifiées pour, par exemple, utiliser la voie avec le moins du trafic. L'exploitation du parallélisme est également affectée par une telle combinaison car celle-ci réduit le nombre d'éléments qui peuvent coopérer simultanément pour exécuter les opérations d'E/S. Les performances globales du système peuvent être ainsi atténuées.

Divers types de connexions sont utilisés entre les éléments fonctionnels lorsque ceux-ci sont indépendants (cf. figure 2.10). Une UC est connectée à un PES par un lien normal d'interconnexion de processeurs. En revanche, pour communiquer avec les contrôleurs, les PES utilisent en général des interfaces de haut débit comme HIPPI (**Hi**gh **P**erformance **P**arallel **I**nterface) [CGK90], IPI-3 (**I**ntelligent **P**eripherals **I**nterface) [Wr89] ou SCSI-2 (**S**mall **C**omputer **S**ystem **I**nterface) [Lo85]. Pour les communications entre les CTRs et les disques, l'interface de bas niveau offerte par le dispositif est utilisée, comme les interfaces SCSI, IPI-2, etc.

Dans le but de limiter les interférences entre le trafic dû aux communications inter-processus et le trafic propre aux E/S, de nouvelles organisations plus flexibles ont été proposées pour les

réseaux d'interconnexion. Un réseau supplémentaire a été ainsi ajouté pour relier les nœuds d'E/S entre eux. De cette façon, toute communication interne entre les nœuds d'E/S ne perturbe pas les communications des processus utilisateurs.

La figure 2.11 montre la structure générale d'une architecture d'E/S pour machines massivement parallèles proposée dans le projet PUMA [Ec91]. Ici, trois réseaux d'interconnexion sont utilisés afin de bien isoler chaque fonction des sous-systèmes. Chaque réseau d'interconnexion permet non seulement de communiquer entre toute paire d'éléments d'un niveau, mais aussi il sert de passerelle pour la communication entre deux niveaux adjacents.

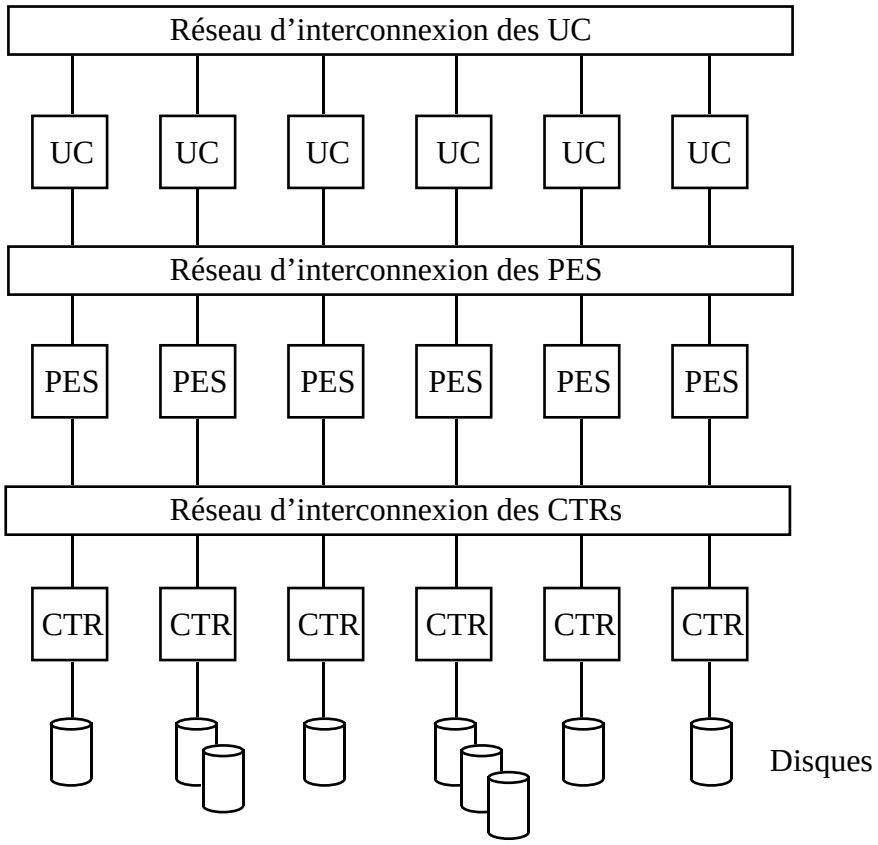


Figure 2.11 Architecture parallèle des E/S.

Dans une telle architecture, quatre niveaux de parallélisme sont possibles. Les trois premiers niveaux sont exploités par le logiciel, mais le quatrième doit l'être par le matériel :

- *Parallélisme des requêtes d'E/S émanants des UC* : toute UC doit être capable de générer ses propres requêtes d'E/S sans avoir à s'adresser à un serveur spécifique de l'application. Chaque UC peut manipuler ses propres fichiers, et la gestion des caches peut être faite à l'intérieur des UC.

- *Serveur parallèle (sur les PES) d'accès aux données* : une application parallèle (gestionnaire de fichiers) peut s'exécuter sur les PES, indépendamment de toute autre activité sur la machine. Un gestionnaire de fichiers ainsi distribué peut répondre en parallèle à plusieurs requêtes en provenance des UC et des CTRs.

- *Accès parallèle aux contrôleurs de disque* : chaque PES peut commander un sous-ensemble des opérations d'E/S indépendamment des autres PES. En effet, l'exécution des opérations d'E/S est distribuée sur tous les PES, et chacun dispose de toutes les informations nécessaires pour traiter la partie qui lui est attribuée.

- *Accès parallèle aux dispositifs de stockage* : le transfert d'un contrôleur vers l'unité de stockage peut être réalisé en parallèle grâce à des disques dotés de plusieurs têtes de lecture ou à des vecteurs de disques (cf. chapitre 3).

Ce type d'architecture dispose de quatre degrés de liberté pour son organisation conformément aux quatre niveaux décrits. Le système peut donc être caractérisé par le quadruple (*p, i, c, d*) où *p* est le nombre d'UC, *i* le nombre de PES, *c* le nombre de contrôleurs et *d* le nombre de disques par contrôleur. La quatrième valeur peut être différente pour chaque CTR mais elle est couramment supposée constante. Cette caractérisation permet aux concepteurs de machines de construire un système qui s'adapte aux besoins des applications et qui est facilement extensible en taille et en performance. L'existence des trois réseaux nécessite cependant des protocoles de communication plus complexes et accroît le coût de conception des machines.

L'avantage de cette architecture est qu'elle assez générale, et de ce fait, elle est non seulement facile d'adapter aux besoins particuliers des utilisateurs, mais aussi, elle peut être facilement simulée, du moins dans sa décomposition logique, par d'autres architectures. Cette architecture appartient à la classe des architectures à disques communs, et le mode de fonctionnement est supposé MIMD avec une mémoire privée par processeur.

2.4 Conclusion

Ce chapitre nous a permis d'introduire le cadre matériel de notre travail. Dans un premier temps nous avons présenté les différentes approches architecturales du parallélisme massif. La caractéristique majeure que nous retiendrons est la flexibilité propre à ces architectures. Ainsi une architecture massivement parallèle s'adapte naturellement aux divers besoins des applications. Toute nouvelle évolution de ce domaine, et en particulier dans les E/S, doit en conséquence conserver cette caractéristique d'extensibilité.

Une fois présentés les concepts architecturaux de base, nous avons décrit le principe général des E/S parallèles. L'efficacité de tels systèmes illustrée, nous avons proposé une première classification des sous-systèmes d'E/S en fonction de leur structure matériel pour accéder aux informations. Notre classification des sous-systèmes d'E/S est volontairement similaire à celle des architectures parallèles, ce qui nous permet de garder une analogie entre les accès à la mémoire vive et les accès à la mémoire secondaire. En effet, bien que le critère de classification

soit le même (accès privé ou partagé aux ressources d'une machine), les implications sont complètement différentes. Alors que pour la mémoire vive on peut parler des données privées et partagées, pour les données sur disque ceci n'est pas tout-à-fait le cas. Les disques sont un moyen de stockage à long terme, et même de coopération entre plusieurs applications à travers les fichiers, et auxquels les utilisateurs veulent accéder sans se soucier de leur localisation physique dans un réseau d'interconnexion. Nous considérons alors que les disques doivent être traités comme des ressources partagées, sans que pour cela il soit exclu l'utilisation des disques privés, mais dans ce cas, ils ne doivent contenir que des fichiers temporaires propres aux processus qui s'exécutent sur le processeur auquel ils sont attachés. L'architecture de test de sous-systèmes d'E/S proposée par M. Henderson et al. [HNS94] au laboratoire Argonne aux Etats-Unis combine en effet l'accès aux disques comme une ressource partagée par tous les processeurs et la possibilité pour ceux derniers d'avoir un disque local utilisé comme cache des disques globaux.

A la fin du chapitre nous nous sommes intéressés à une architecture générale pour les E/S. Cette architecture donne lieu à quatre possibilités de parallélisme dont trois sont à exploiter par le logiciel de traitement des E/S et par les mécanismes système sur lesquels ce logiciel doit être basé. Nous reviendrons sur cette structure pour l'étude de l'architecture logiciel des sous-systèmes parallèles d'E/S.

Chapitre 3
Les unités de disques

Au coeur des sous systèmes d'E/S, les unités de disques ont suscité l'intérêt de la communauté scientifique soucieuse des performances des E/S. Dans le chapitre précédent nous avons vu que le principe des E/S parallèles s'appuie sur l'utilisation efficace de plusieurs de ces unités pour obtenir une représentation distribuée des fichiers. Outre la technologie des dispositifs d'E/S en tant qu'unités indépendantes, il s'avère nécessaire d'étudier plus en détail le regroupement de ces dispositifs pour le stockage d'informations distribuées. C'est l'objet de ce chapitre.

La course aux performances ne doit en aucun cas sacrifier l'équilibre entre toutes les composantes d'un système informatique. Ainsi, le stockage en mémoire secondaire doit évoluer dans les mêmes proportions que toute autre composante. Pour quantifier les progrès de la technologie des disques magnétiques, on mesure couramment la densité atteinte (ou MAD pour maximal area density), qui représente le nombre de bits stockés par pouce carré. Les travaux de P.D. Frank [Fr87] ont permis de dégager une "première loi de densité des disques", basée sur les évolutions de la valeur de MAD les deux dernières décennies :

$$MAD \cong 10^{(Année-1971)/10}$$

L'application de cette loi reste aujourd'hui valable si l'on considère celle-ci comme une moyenne, car dans les toutes dernières années la technologie des disques modernes dépasse légèrement les prédictions de P.D. Franck. En effet, nous avons constaté que pour certains disques comme le Seagate ST-5850A, la valeur de MAD atteinte en 1994 était de 235 Méga-bits par pouce carré alors que la première loi de densité des disques ne prévoyait que 200. Pour cette année l'écart devrait encore se creuser car les fabricants de disques sortent déjà au marché des unités qui arrivent à stocker jusqu'à 295 Méga-bits dans un pouce carré (disque ST-9655AG) ; si l'on calcule la valeur de MAD prédite, elle n'est que de 252.

Outre cette loi, on a observé depuis les années 70 que la technologie des disques magnétiques a doublé sa capacité et réduit de moitié ses prix tous les trois ans. Ces améliorations peuvent être

considérées comme acceptables mais surtout elles sont tout à fait comparables à celles d'autres parties des systèmes informatiques. Plus particulièrement il est à noter que les mémoires vives et les mémoires secondaires ont conservé, en matière de densité de stockage et de prix, et au regard de leurs progrès respectifs, une corrélation étroite.

Cependant les performances globales d'un sous-système d'E/S ne dépendent pas uniquement des densités de stockage ; la rapidité à laquelle les informations sont délivrées aux unités de calcul détermine en dernière instance les performances du sous-système. Pour la mémoire vive cette vitesse a augmenté fortement grâce à l'apparition des mémoires caches et grâce aux progrès propres de la technologie des circuits intégrés. Pour les dispositifs d'E/S les progrès en ce domaine sont beaucoup plus modestes. Un disque magnétique est un dispositif mécanique dont la vitesse de transfert est conditionnée principalement par le déplacement des têtes de lecture/écriture (positionnement) et par la rotation du disque. Dans les 10 dernières années le temps de positionnement et le temps de rotation (temps nécessaire pour une rotation de 360°) se sont réduits seulement d'un facteur de 2.

Pour pallier à ce problème une solution souvent proposée consiste à utiliser la commutation de circuits comme principe de stockage pour les mémoires de masse. Cette technique, qui n'exige aucun mouvement, permet d'accélérer nettement le débit du transfert disque - unité de calcul [KGP89] et d'atteindre des performances remarquables, de 2 Giga octets/s, ce qui est à peu près 700 fois plus rapide qu'un dispositif d'E/S standard. Cependant, le coût de telles mémoires est prohibitif, leur capacité de stockage est limitée à quelques Giga octets et de plus, l'information qui y est mémorisée reste volatile. Utilisées seulement dans des super-calculateurs équipés d'un matériel spécialisé elles ne connaissent de ce fait qu'un essor commercial très restreint.

Les disques magnétiques restent donc la brique de base incontournable à la conception de sous-systèmes d'E/S parallèles. Ces dispositifs sont amplement répandus, faciles à utiliser et représentent une méthode économique pour stocker de grands volumes d'information.

Centrées principalement sur la célérité des opérations de lecture et d'écriture, de nombreuses tentatives ont été entreprises pour augmenter la vitesse des dispositifs d'E/S standards. Bien qu'il soit désormais possible, à l'aide de semi-conducteurs, d'obtenir des débits de transfert proches de 50 Mbits/s, ceci ne représente que le double de la vitesse d'un disque classique.

En résumé, tandis que les processeurs doublent chaque année leur puissance de calcul, le temps de positionnement des disques n'est divisé par deux que tous les dix ans. Ce déséquilibre est très net lorsque l'on observe l'évolution des composantes informatiques essentielles ; la figure 3.1 illustre ces différences. Il est admis que cette situation demeurera et que les contraintes

mécaniques des dispositifs d'E/S ne suivront pas les développements technologiques des unités de calcul.

Dans la suite de ce chapitre nous présentons les diverses techniques proposées pour améliorer les performances des dispositifs d'E/S, et plus spécialement les vecteurs de disques car ils représentent aujourd'hui le choix des concepteurs des sous-systèmes d'E/S à hautes performances. Mais avant cela nous commençons, dans un souci de clarté, par analyser la structure des disques magnétiques et dégageons les facteurs qui déterminent leurs performances.

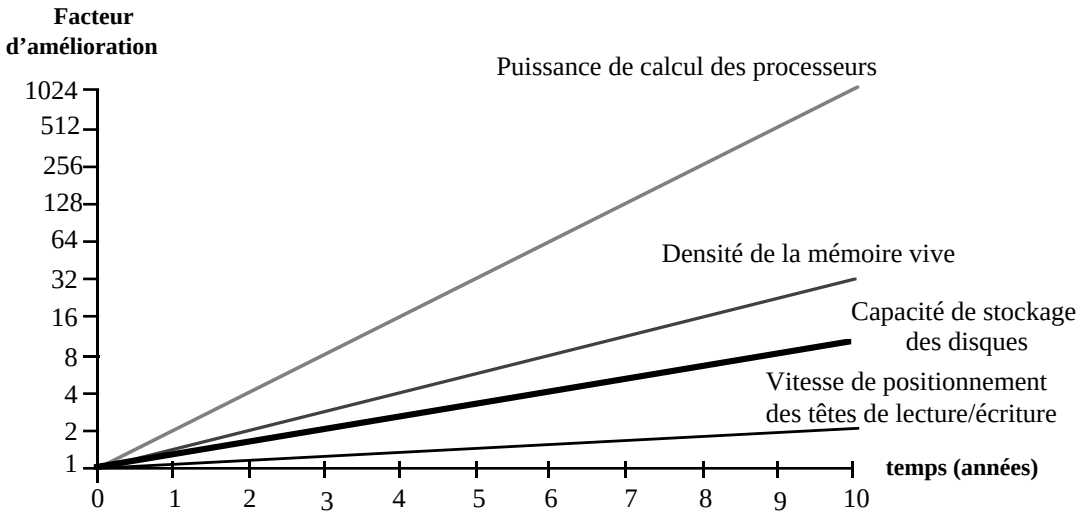


Figure 3.1 Différences dans l'évolution des composantes informatiques.

3.1 Architecture générale des disques magnétiques

Un disque magnétique, dont la structure est détaillée dans la figure 3.2, est composé en fait d'une *pile de disques* métalliques recouverts sur leurs deux faces d'un matériau magnétique. Chacune de ces faces est divisée, tout d'abord en *pistes* de l'axe de rotation vers l'extérieur, puis en *secteurs* selon un angle constant. Une piste est donc un enregistrement circulaire composé de secteurs, et un secteur est l'unité de lecture/écriture dans une piste (tous les bits qu'il contient sont lus ou écrits simultanément). Le nombre de pistes par face et de secteurs par piste est constant pour toute la pile. De plus l'angle constant et régulier de découpage des pistes en secteurs impose que, pour maintenir d'une piste à autre une vitesse de rotation constante lors d'une opération, la densité exploitée des secteurs varie d'une piste à autre. Plus la piste est proche de l'axe, plus la densité exploitée est élevée. Ceci permet que les disques tournent à une vitesse constante et que la même quantité d'information soit stockée dans les pistes externes et

internes. Les disques les plus modernes exploitent au mieux la densité maximum pour chaque piste ; donc plus la piste est externe, plus ses secteurs contiennent d'information. La capacité de stockage est ainsi maximale, mais cela nécessite toutefois une électronique plus sophistiquée pour les opérations de lecture et d'écriture.

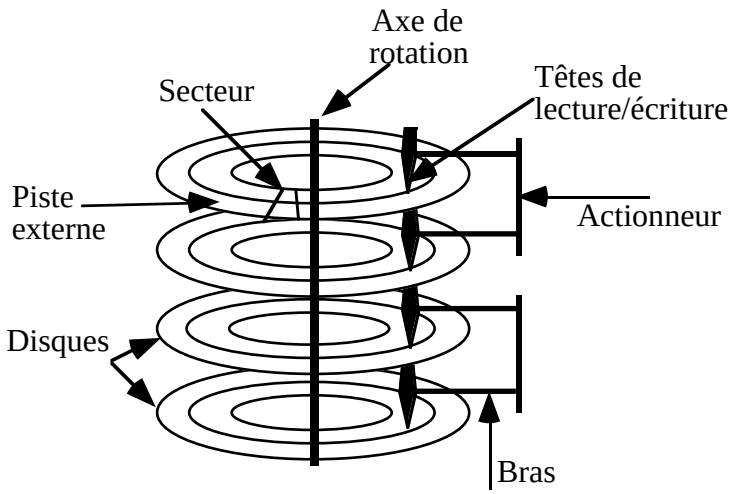


Figure 3.2 Structure d'une unité de disque magnétique.

Les *têtes* de lecture/écriture sont des électro-aimants qui produisent des champs magnétiques commutables pour écrire ou détecter des chaînes de bits sur une piste. Une tête de lecture/écriture n'est jamais en contact avec la surface du disque mais elle la "survole" simplement. Un dispositif mécanique, *l'actionneur*, contrôle le mouvement des *bras* qui positionnent les têtes électroniques au-dessus des pistes.

Dans la pile de disques, l'ensemble de pistes (1 par face) superposées parallèles, c'est-à-dire accessibles simultanément par une même position des bras, forme un *cylindre*. Le nombre de secteurs enregistrables sans nouveau mouvement des bras est égal au nombre de secteurs par piste fois le nombre de faces d'un cylindre fois (le cas échéant) le nombre de têtes par bras et par face (souvent 1).

Les unités de disques sont reliées au reste du système par des *interfaces*. Diverses interfaces propriétaires et standards ont été définies pour transférer l'information entre l'unité centrale et les disques. Ces interfaces définissent la façon d'extraire les impulsions des champs magnétiques, le codage de ces impulsions en bits et leur alignement, des codes d'erreur et d'autres informations nécessaires pour générer les paquets prêts à être livrés aux unités de calcul. Comme nous l'avons vu au chapitre 2, SCSI [Lo85], IPI [Wr89] et plus récemment HIPPI [CGK90], sont les standards les plus répandus dans l'industrie. D'autres interfaces qui méritent

d'être citées sont ST506, SMD (**S**torage **M**odule **I**nterface) et ESDI (**E**nhanced **S**mall **D**evice **I**nterface) [KGP89].

Le dernier volet de l'architecture des unités de disques à considérer est la structure de connexion entre la mémoire vive et le(s) disque(s). C'est la réplication des voies de connexion qui nous intéresse plus particulièrement car elle conditionne une meilleure exploitation du parallélisme d'un système d'E/S. La structure de connexion varie fortement en fonction du niveau de performance exigé. Une station de travail utilisera une interface SCSI pour connecter directement les deux éléments, tandis qu'un ordinateur plus puissant aura un contrôleur de disque pour gérer plusieurs unités de disques en même temps. Enfin, un ordinateur central (*mainframe*) disposera de plusieurs unités et des schémas d'interconnexion plus complexes. Ces systèmes utilisent les *canaux d'E/S* qui sont munis de processeurs spécialisés capables d'exécuter un programme de lecture/écriture installé par l'unité centrale sur la mémoire vive de la machine.

3.2 Facteurs déterminants pour les performances d'un disque

Avant de présenter les extensions apportées aux architectures de disques magnétiques, nous considérons important de bien comprendre les éléments qui influent sur les performances des unités de disques. Les performances d'un disque sont fonction du temps de service de l'unité. Trois facteurs sont déterminants pour calculer ce temps de service : le temps de positionnement (*seek time*), la latence de rotation (*rotational latency*) et le temps effectif de transfert (*transfer time*)[1].

Le temps de positionnement est le temps nécessaire pour déplacer les têtes de lecture/écriture au-dessus de la piste où se trouve l'information qui doit être transférée. Il est égal au temps de démarrage de l'actionneur (environ 6ms) augmenté du temps de traversé des pistes qui séparent la position à atteindre de la position actuelle des têtes. En fonction du nombre de pistes du disque ce temps de positionnement varie en moyenne de 7 à 20 ms [CLGK94].

La latence de rotation est définie par le temps qui s'écoule avant que le secteur à lire ou à écrire ne passe au-dessous de la tête de lecture/écriture, une fois celle-ci est positionnée sur la bonne piste. Si nous considérons que la vitesse de rotation est d'approximativement 6000 tours/minute, le temps maximum de rotation serait de 10 ms avec un temps moyen de 5 ms.

[1] Dans un système surchargé l'attente pour la libération d'un disque peut être un facteur important dans le calcul de ce temps de service.

Finalement, le temps de transfert est le temps requis pour réaliser le transfert effectif des octets demandés entre le disque et la mémoire. Celui-ci est le seul paramètre qui dépend du nombre d'octets à transférer ; le temps de positionnement et le temps de rotation restent les mêmes quelle que soit la taille des données. Le transfert de grands blocs de données est alors mieux amorti que celui de petits paquets d'informations. Le débit de transfert de données des disques d'aujourd'hui se situe entre 3 et 4 Méga-octets/s.

Un sous-système d'E/S qui se veut performant doit minimiser chaque composante du temps de service. Si l'on suppose de petites tailles pour les données à transférer, le temps de transfert est une partie négligeable du temps global ; et les voies de connexion pourraient être utilisées pour transférer en même temps des donnés de différentes opérations d'E/S. Un tel sous-système peut effectuer un positionnement tout en transférant un bloc de données. De cette façon la bande passante est augmentée et le temps d'attente de libération du dispositif est réduit.

3.3 Vers les E/S à hautes performances

Dans cette section nous commençons tout d'abord par la présentation des techniques classiques proposées pour améliorer les performances des dispositifs d'E/S. L'efficacité de ces solutions est fortement dépendante des applications qui s'exécutent sur la machine, ce qui explique que leur utilisation reste restreint à des domaines bien spécifiques [KGP89]. Aujourd'hui, le regroupement de plusieurs disques pour former une seule unité logique est une technique amplement répandue dans les systèmes à hautes performances. Cette solution et l'organisation des données qui y sont stockées sont étudiées à la fin de la section.

3.3.1 Les solutions classiques

La réduction du temps de service diminue automatiquement les temps d'attente des applications et en conséquence les performances globales des systèmes sont améliorées. Les différentes approches existantes pour réduire ce temps de service portent sur un ou plusieurs des facteurs étudiés dans la section précédente. Les techniques présentées sont : les disques à têtes fixes, l'augmentation de la densité de stockage, les disques à commutation électronique de circuits, les caches de disques et l'ordonnancement des requêtes d'E/S.

3.3.1.1 Les disques à têtes fixes

L'idée est simple : on place une tête de lecture/écriture sur chaque piste du disque. Le besoin de positionner les têtes est ainsi éliminé et par conséquent le temps de positionnement est nul. Cette solution fut utilisée comme mémoire d'arrière-plan pour la mémoire virtuelle, mais il a cessé de servir lors de l'apparition des disques à plusieurs centaines de pistes, ce qui posait une barrière économique trop importante pour son implémentation.

3.3.1.2 Transfert en parallèle à partir de la pile de disques

Il existe aujourd'hui des dispositifs disques capables de lire ou écrire en même temps sur plusieurs surfaces dans une pile. Ceci augmente le débit de transfert mais n'a aucune incidence sur le temps de positionnement ni sur la latence de rotation. Cette solution, bien qu'utilisée par différents constructeurs, est coûteuse car pour sa réalisation la présence de plusieurs actionneurs est indispensable.

3.3.1.3 Augmenter la densité des disques

La toute nouvelle technologie des têtes magnétorésistives conçue par Hewlett Packard et les mécanismes de lecture de données PRML (**P**artial **R**esponse **M**aximum **L**ikelihood) devraient dominer le développement des futures générations de disques magnétiques. Parce que ces deux technologies améliorent les capacités de lecture des bits dans des contextes dégradés, elles autorisent l'enregistrement de fortes densités de données. D'après un communiqué de presse de Hewlett Packard, la capacité des disques magnétiques devraient ainsi pouvoir être multipliée par dix dans la décennie à venir, avec des densités qui atteindront 2 Giga-bits par pouce carré.

Augmenter la densité des disques fait diminuer le temps de transfert puisque plus de bits sont lus/écrits par unité de temps. Néanmoins cette solution ne concerne ni la latence de rotation ni le temps de positionnement. Elle est par contre adéquate pour des systèmes où les E/S manipulent de grands volumes d'information à chaque opération.

3.3.1.4 Disques à commutation électronique[1]

Les disques à commutation électronique sont construits à partir de puces lentes de mémoire et peuvent être vus comme une mémoire lente ou comme un disque à haute vitesse. En raison de leur rapidité par rapport aux disques classiques, les opérations d'E/S qui y ont lieu n'ont pas

[1] Solid State Disks

besoin d'être asynchrones avec l'unité de calcul. Le processus qui exécute une opération d'E/S récupère les données directement du disque, ce qui élimine le surcoût du système d'exploitation pour gérer les opérations d'E/S. Ils ont aussi été utilisés comme mémoire d'arrière-plan.

La stabilité des informations qui sont stockées sur ce type d'unités est garantie par des batteries externes, cependant le risque de perte de l'information n'est pas négligeable. Si les batteries ne sont pas chargées lors d'une coupure du courant toute l'information est perdue. C'est pourquoi ces disques sont utilisés plutôt pour des données temporaires que pour des données permanentes.

Un autre problème des disques à commutation électronique est leur coût : 1 Mbit stocké sur un de ces disques est à peu près 15 fois plus cher que le même Mbit stocké sur un disque magnétique. Ceci fait que leur utilisation ne se soit pas étendue aux domaines au-delà des super-calculateurs des grands centres de calcul.

3.3.1.5 Les caches de disque

Les caches de disque sont simplement des anté-mémoires qui servent de tampons entre la mémoire principale et les disques. A chaque accès à un bloc de données en disque, on garde une copie de celui-ci dans un tampon du système, et toute nouvelle référence à ce bloc sera résolue grâce à la copie stockée dans le tampon. Si une information sur disque est référencée à nouveau, ces tampons peuvent effectivement éliminer le temps de positionnement et la latence de rotation. L'efficacité réelle de cette solution est donc fortement dépendante de l'enchaînement des accès aux données, et donc des applications.

Les caches de disques peuvent devenir très utiles lorsqu'ils utilisent des techniques de batteries externes comme celle des disques à commutation électronique. Un cache non volatil permettrait des écritures rapides, car le risque de perte des modifications lors d'une défaillance du système serait écarté (bien entendu les batteries doivent être fiables).

Les caches d'E/S sont une bonne solution pour les machines monoprocesseur, c'est pourquoi leur gestion est aujourd'hui un sujet d'actives recherches. Dans un contexte parallèle et massivement parallèle, il est évident que de tels systèmes sont à envisager, mais leur architecture est à revoir pour les adapter aux caractéristiques spécifiques de ces systèmes. Les chapitres 5 et 7 étudient plus en détail cet aspect fondamental des E/S.

3.3.1.6 Ordonnancement des requêtes

Les retards dus aux mouvements mécaniques des unités peuvent être minimisés avec un bon ordonnancement des requêtes [JSWB93]. Un algorithme qui ordonne la liste de requêtes avec le critère du “plus petit temps de positionnement en premier” réduit le temps de positionnement global. Des algorithmes distribués d’ordonnancement ont été aussi proposés [DJT94]. Cependant pour que l’ordonnancement soit le plus efficace il faut de longues listes de requêtes, et le but des E/S à hautes performances est justement de réduire la longueur de ces listes. Les systèmes d’E/S modernes évitent donc d’avoir recours à cette solution.

3.3.2 Les vecteurs de disques

La croissance de la capacité de stockage des unités de disques pour les gros ordinateurs n’a pas retenu toute l’attention des concepteurs. Les ordinateurs personnels ont créé un marché pour des disques moins chers, de la part d’un public très exigeant. Ces disques ont une capacité de stockage réduite et leurs performances sont moins impressionnantes que celles des disques plus sophistiqués mais leur prix est abordable pour le grand public.

Dans [PGK88], D. Patterson, G. Gibson et R. Katz comparent trois systèmes de disques : l’IBM 3380, le Fujitsu M2361A “Super Eagle” et le Conners CP3100. Ils correspondent à trois différents types de machines, respectivement : les gros ordinateurs, les ordinateurs de taille moyenne et les ordinateurs personnels. La figure 3.3 est une version abrégée de cette comparaison. Les disques plus chers ont évidement des capacités de stockage et des performances supérieures à celles des disques économiques ; en revanche, ce qui est beaucoup plus surprenant, c’est que le débit par actionneur des gros disques ne soit même pas le double du débit d’un disque économique. Sur la plupart des autres paramètres, les valeurs affichées pour les disques économiques sont supérieures ou au moins égales (en qualité) à celles des gros disques.

La petite taille et la basse consommation des disques économiques sont loin d’être des facteurs négligeables lorsqu’il s’agit de la conception des systèmes massivement parallèles, surtout si l’on considère que ces disques ont déjà intégré dans l’unité la plupart de fonctions des contrôleurs des grosses machines. Ceci est rendu possible grâce aux efforts des comités de standardisation pour définir des interfaces de haut niveau pour la connexion des périphériques aux bus de communication (SCSI par exemple). L’adoption de ces standards a permis leur implémentation matériel à des prix réduits, et leur intégration aux unités de disques de petite taille.

Caractéristiques	IBM 3380	Fujitsu M2361A	Conners CP3100
Capacité (Mo)	7500	600	100
Prix/Mo	$18-10	$20-17	$10-7
MTTF prévu (heures)	30.000	20.000	30.000
MTTF[1] observé (heures)	100.000	-	-
Nombre d'actionneurs	4	1	1
Opérations d'E/S/s par actionneur (maximum)	50	40	30
Opération d'E/S/s par actionneur (observées)	30	24	20
Débit de transfert (Mo/s)	3	2,5	1
Consommation (W)	6.600	640	10
Volume (pieds^3)	24	3,4	0,03

Figure 3.3 Comparaison des différentes unités de disques.

D'après cette comparaison, un ensemble de disques économiques atteindrait donc la même capacité de stockage d'un disque plus onéreux, consommerait bien moins et requerrait un volume beaucoup plus réduit. En général, on peut dire que les disques économiques se comportent au moins aussi bien que ceux de grande capacité.

Toutes ces caractéristiques nous amènent aux propositions faites au milieu des années 80 qui préconisaient la construction des sous-systèmes d'E/S de grande capacité à hautes performances par le regroupement de plusieurs disques économiques [Ki86, SG86]. Ces regroupements ont reçu le nom de vecteurs de disques (*disk arrays*). Comme le montre la figure 3.3, un vecteur de 75 disques CP3100 a la même capacité de stockage qu'un IBM 3380, avec potentiellement 12 fois sa bande passante en opérations d'E/S (75*20*1 vs. 30*4). Ceci à un coût moindre et pour une consommation et un espace physique nettement inférieurs.

Le point faible de cette solution est la fiabilité du sous-système d'E/S. Le risque d'une panne augmente car les probabilités de défaillances des disques à l'intérieur d'un vecteur sont

[1] Pour Mean Time To Failure, soit le temps moyen de fonctionnement sans panne

indépendantes. Le temps moyen avant qu'un disque ne tombe en panne (MTTF) est un paramètre de sûreté pour l'information qui y est stockée. Si l'on suppose un taux de défaillance constant, c'est-à-dire, que le temps entre deux pannes suit une loi exponentielle, et que la panne d'une unité est un événement indépendant, nous pouvons calculer le MTTF d'un vecteur de disques comme :

$$\text{MTTF(vecteur de disques)} = \frac{\text{MTTF d'un seul disque}}{\text{Nombre de disques dans le vecteur}}$$

Toujours à partir de l'information de la figure 3.3, la valeur du MTTF pour le vecteur de 75 disques est de 300 heures, soit moins de 2 semaines. De toute évidence un système où l'un de ses composants risque de tomber en panne toutes les 2 semaines ne peut pas concurrencer le disque IBM originel avec une moyenne de service de 30.000 heures, soit plus de 3 ans avant qu'il ne présente un problème. Les vecteurs de disques ne sont donc pas une solution très réaliste, à moins qu'un effort considérable ne soit fait dans le domaine de la tolérance aux pannes.

Les systèmes RAID (**R**edundant **A**rray of **I**nexpensive **D**isks, ou vecteur redondant de disques économiques) proposés par D. Patterson, G. Gibson et R. Katz dans [PGK88], viennent combler cette lacune. Grâce à la redondance de l'information, ces systèmes peuvent offrir une disponibilité de service comparable, voire supérieure, à celle des disques indépendants. Plusieurs organisations différentes des systèmes RAID existent aujourd'hui et afin de mieux les comprendre nous présentons tout d'abord les différentes taxonomies possibles des vecteurs de disques.

3.3.3 Organisations des données dans un vecteur de disques

Les *fichiers entrelacés* sont à l'origine de toute organisation de données dans un vecteur de disques. Un fichier entrelacé est un fichier dont les blocs sont distribués sur plusieurs disques, ce qui donne une organisation qui ressemble à un tableau à deux dimensions (figure 3.4). Chaque colonne du tableau est stockée sur un disque différent selon, typiquement, la politique du tourniquet. Si le premier bloc est stocké sur le disque k, et si p est le nombre total de disques utilisés, alors, selon cette politique, le n-ième bloc est stocké sur le disque $((n + k) \bmod p)$.

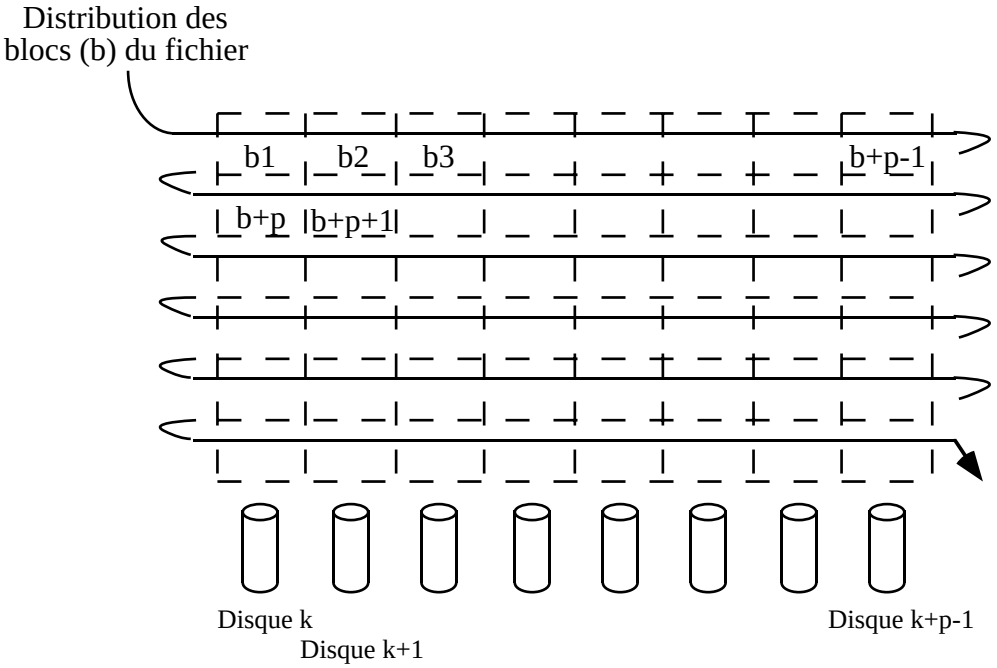


Figure 3.4 Concept des fichiers entrelacés.

Dans [Cr89], T. Crockett étudie l'organisation des fichiers dans un environnement parallèle. Il identifie ainsi plusieurs types d'accès aux fichiers et conclu qu'aucune organisation de données sur disque n'est optimale pour tous les types d'accès observés. La distribution de fichiers s'avère néanmoins efficace pour la plupart des cas, mais la participation de l'utilisateur pour déterminer les conditions de cette distribution reste indispensable pour mieux adapter l'organisation physique à la structure logique des fichiers. La comparaison de ce type de systèmes effectuée par G.R. Ganger et al. [GWHP93] confirme cette conclusion. Quatre systèmes de vecteurs de disques peuvent être identifiés selon l'organisation des données implémentée : les systèmes traditionnels, les disques synchrones, les vecteurs à entrelacement des fichiers, et les systèmes synchrones à fichiers entrelacés. Nous présentons à continuation le principe de chaque système.

3.3.3.1 Système traditionnel

Dans ce type de systèmes il n'y a pas d'entrelacement des fichiers, et chaque fichier est entièrement stocké sur un seul disque ; le disque est donc une unité indépendante. Simple à implanter, un tel système atteint un bon niveau de performances lorsque les requêtes partent sur des blocs de données distincts correctement distribués sur les disques. En revanche dans le cas où les blocs accédés ne sont pas équitablement distribués, les performances peuvent fortement se dégrader.

Comme aucun des trois facteurs de performances étudiés (temps de positionnement, temps de rotation et temps de transfert) n'est amélioré, le temps de service d'un vecteur de disques, avec cette technique d'organisation des données, est le même que si le stockage des données se faisait sur un seul grand disque. Si toutefois il est possible d'observer une amélioration des performances, ce n'est que grâce à la réduction de la taille de la liste d'attente dans laquelle doit séjourner toute requête avant d'être traitée.

3.3.3.2 Disques synchrones

Dans une organisation de disques synchrones les fichiers sont entrelacés entre eux et un par un, octet par octet sur chaque disque [Ki86]. Cette technique, très utile lorsque le temps de transfert influe beaucoup sur le temps de service des E/S, impose une synchronisation de tous les disques, de telle sorte que les têtes de lecture/écriture soient positionnées sur le même secteur sur tous les disques. Les disques fonctionnent alors ensemble, à l'image d'un gros disque qui aurait un débit de transfert et une capacité d'autant plus accrus que le nombre de disques du vecteur est important. Le système est équilibré par rapport au nombre de requêtes car chaque requête est traitée simultanément par tous les disques. Puisque le temps de positionnement et la latence de rotation ne sont pas modifiés, cette organisation conduit à de bons résultats seulement lorsque le temps de transfert domine sur les autres facteurs du temps de service.

Cette technique ne nécessite pas forcément une synchronisation de tous les disques du vecteur, mais plutôt une synchronisation partielle par unités. Les disques synchronisés forment toujours une unité est leur nombre est appelé *degré de synchronisation (ds).* Par exemple, un vecteur de 16 disques avec un degré de synchronisation 2 est un système formé par 2 unités synchrones de 8 disques chacune. La valeur de *ds* peut varier de 1 au nombre de disques du système ; dans ce dernier cas on parle alors des systèmes *complètement synchrones*.

3.3.3.3 Entrelacement de fichiers par blocs

Cette technique diffère des précédentes, non seulement parce qu'il n'y a pas de synchronisation entre les disques, mais aussi par rapport à la taille des unités de morcellement et d'entrelacement des fichiers ; ici, l'unité de découpage est le bloc, au sens traditionnel du terme.

Avec cette organisation des données, la lecture ou l'écriture en parallèle de plusieurs blocs, d'un même fichier, sur des disques différents est autorisée. Lorsqu'une requête est transmise au système de disques, deux cas se distinguent selon que la requête concerne un seul bloc ou plusieurs blocs distincts. Dans le premier cas, comme un seul bloc est concerné, la requête est simplement déposée dans la liste d'attente du disque correspondant. Si au contraire, la requête

nécessite le transfert de *m* blocs, elle est décomposée en *m* requêtes, une par bloc, et chacune d'elles est mise en attente dans la liste du disque associé au bloc qu'elle représente. Dans ce cas le résultat est un accès en parallèle à l'information mais pas synchrone pour autant. Ceci est également valable pour plusieurs requêtes indépendantes d'un seul bloc chacune. Dans tous les cas le système cherche à traiter le plus possible d'opérations simples d'E/S en même temps ; l'idéal est de traiter une opération par disque, c'est-à-dire par file d'attente.

Vis-à-vis de la charge des disques, l'avantage d'une telle organisation est qu'elle conserve l'équité entre les disques : les données et les requêtes sont correctement répartis entre eux. Tous les disques reçoivent à peu près le même nombre de requêtes. Bien qu'à la différence d'un système traditionnel, dans les systèmes à entrelacement de fichiers il faille payer le coût d'un temps de positionnement et d'une latence de rotation pour chaque bloc, et non pas pour chaque requête, le temps de service n'est pas pour autant augmenté car les traitements sont effectués en parallèle (cf. section 2.3).

Le nombre d'accès simultanés autorisés pour un fichier est appelé *degré d'entrelacement (de)*. Dans la plupart des réalisations, la valeur de *de* est égale au nombre de disques du vecteur, mais d'autres valeurs sont tout à fait envisageables.

3.3.3.4 Les systèmes synchrones à fichiers entrelacés par blocs

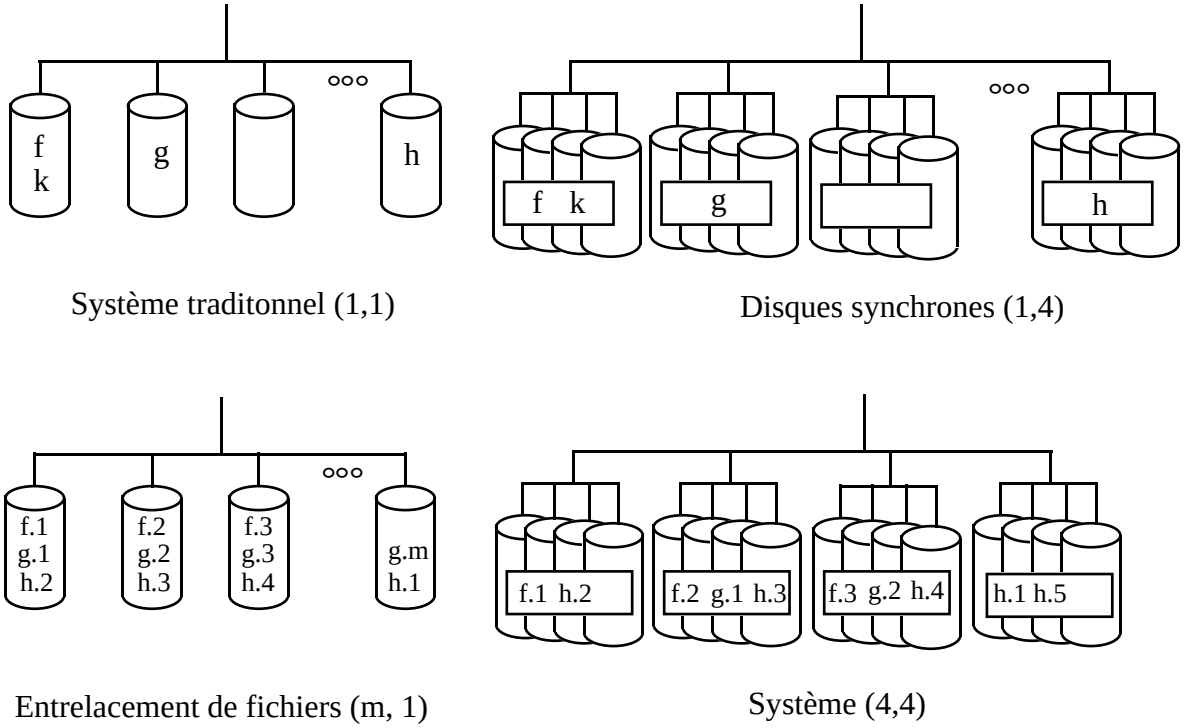
Une combinaison des deux dernières techniques est également possible si l'on entrelace les fichiers par blocs sur les différentes unités synchrones. Le nombre maximum de disques qui peuvent être ainsi synchronisés est limité, et dans certains cas le découpage de l'information sur les unités synchrones peut augmenter les performances du système.

En dépit du fait qu'ils aient comme même objectif la réduction du temps de service des requêtes d'E/S, la synchronisation des disques et l'entrelacement des fichiers sont fondamentalement différents. Avec les systèmes synchrones un fichier est entièrement stocké sur un disque, d'où l'impossibilité d'accéder simultanément à divers blocs du même fichier. Ils glanent leur profit grâce au principe de localité des références à un fichier, puisque le transfert d'une grande quantité d'information ne fait intervenir qu'une seule fois le temps de positionnement et la latence de rotation. En revanche, la solution alternative des fichiers entrelacés, si elle néglige ce principe de localité, décompose l'accès à un grand volume d'informations en plusieurs accès à des blocs plus petits, ce qui permet la lecture/écriture en parallèle de plusieurs blocs d'un même fichier. Nous pouvons dire que les disques synchrones sont aux disques à fichiers entrelacés, ce que le modèle architectural SIMD est au modèle MIMD des architectures

parallèles. L'organisation synchrone à fichiers entrelacés par blocs représente alors l'approche MSIMD des architectures parallèles.

La figure 3.5 résume les différentes organisations des vecteurs de disques. Les systèmes sont caractérisés par le couple (*de, ds*). Si *m* est le nombre total de disques du système, un système traditionnel est alors un système (1,1), un système à entrelacement total est (*m,* 1) et un système totalement synchronisé est (1, *m*). Il est évident que $de \leq m/ds$, c'est-à-dire que le degré d'entrelacement est limité par le nombre d'unités indépendantes.

f : fichier entier f
f.x : bloc x du fichier f



Système traditonnel (1,1)

Disques synchrones (1,4)

Entrelacement de fichiers (m, 1)

Système (4,4)

Figure 3.5 Les organisations des vecteurs de disques.

Dans [BN89], A.L. Narasimha Reddy et P. Banerjee font une évaluation des quatre types d'organisation des données présentées. Il a été observé que les systèmes synchrones affichent les meilleures performances lorsque la charge du sous-système d'E/S n'est pas très importante ; ceci est dû à la surcharge nécessaire pour gérer le parallélisme dans les systèmes entrelacés. Cependant, la taille de l'unité de transfert entre les disques est la mémoire s'est révélée déterminante dans les performances des opérations. Ainsi, des opérations d'E/S qui nécessitent de transferts de données de grande taille sont avantagées par les systèmes synchrones alors que les systèmes asynchrones répondent mieux lorsqu'il s'agit d'un très grand nombre de requêtes concernant des données de petite taille.

Avant de passer à la présentation des niveaux RAID, il faut que nous disions un mot sur l'extensibilité de ces systèmes. D'après A.L. Narasimha Reddy et P. Banerjee la synchronisation de disques n'est pas possible au-delà de 8 ou 16 disques par unité synchrone, ce qui élimine l'utilisation des systèmes complètement synchrones dans la conception des systèmes massivement parallèles. De plus, pour obtenir un haut degré de concurrence dans l'accès aux blocs d'un fichier, l'entrelacement d'un fichier par blocs s'avère nécessaire. Le choix entre un système à fichiers entrelacés par blocs avec ou sans unités synchrones reste à déterminer en fonction des types d'accès exécutés par les applications.

3.4 Les systèmes RAID

Les vecteurs redondants de disques économiques cherchent à offrir une haute puissance dans les E/S tout en garantissant une bonne fiabilité. Leur principe est de diviser les vecteurs de disques en groupes fiables et d'associer à chacun de ces groupes, des disques de vérification avec des informations redondantes.

Un système RAID est caractérisé par les paramètres suivants :

Le nombre total D de disques de données (sans les disques de vérification) ;
Le nombre G de disques de données dans un groupe (sans les disques de vérification) ;
Le nombre C de disques de vérification dans un groupe ;
Le nombre $N_G = D/G$ de groupes.

A partir de ces valeurs nous pouvons mesurer l'efficacité de chacune des organisations RAID existantes (appelées couramment niveaux RAID). Nous nous intéressons particulièrement à la fiabilité du système (MTTF), aux performances obtenues par le vecteur et au surcoût qui doit être payé pour assurer cette fiabilité, c'est-à-dire, à l'espace disque utilisé pour le stockage des informations redondantes.

Nous avons vu que le point faible des vecteurs de disques était leur MTTF très court. Pour calculer la valeur du MTTF d'un système RAID, il faut prendre en compte le temps moyen de remplacement d'un disque lors d'une panne, ou MTTR (**M**ean **T**ime **T**o **R**epair), c'est à dire le temps nécessaire pour que l'information qui était contenue dans le disque endommagé soit de nouveau disponible. Il est à noter que ce type de système perdrait sa fiabilité si la défaillance d'un deuxième disque intervenait avant le remplacement de la première unité endommagée, d'où l'importance d'un MTTR borné. Une solution pour minimiser la valeur du MTTR est de toujours disposer d'un disque de rechange, et ainsi procéder à la récupération de l'information

immédiatement après la panne. Basés sur le MTTF d'un disque et sur la probabilité qu'une deuxième panne arrive avant que la première ne soit réparée, D. Patterson, G. Gibson et R.H. Katz [PGK88] proposent la formule suivante pour le calcul du MTTF d'un système RAID :

$$MTTF_{RAID} = \frac{\left(MTTF_{Disque}\right)^2}{\left(D + C + N_G\right) \times \left(G + C - 1\right) \times MTTR}$$

Nous pouvons à présent calculer la valeur du MTTF d'un système RAID. Prenons par exemple un système à 100 disques de données (D=100), 10 disques de données par groupe (G=10 et N_G=10), une durée de vie de 30.000 heures qui correspond à la spécification du MTTF pour un disque standard ($MTTF_{Disque}$=30.000), et un temps moyen de remplacement de 1 heure (MTTR=1). Le nombre de disques de vérification par groupe est déterminé par chaque organisation RAID ; il est compris entre 1 et le nombre de disques de données du groupe ($1 \leq C \leq G$). Avec ces valeurs, nous obtenons une valeur du MTTF supérieure à 40 ans pour toutes les organisations RAID proposées.

3.4.1 Les différentes organisations RAID

La grande fiabilité des organisations RAID diminue l'importance de la valeur du MTTF de chaque organisation. En effet, quelle différence peut avoir entre une organisation avec un MTTF de 40 ans et une autre avec un MTTF de 500 ans ? Dans les deux cas de figure, les systèmes seront sûrement remplacés avant qu'ils n'atteignent leur durée prévue de vie. Nous orientons alors notre présentation vers les critères qui font les différences entre les différentes organisations, à savoir, le temps de réponse aux requêtes d'E/S et la capacité de stockage effectivement utilisée, c'est-à-dire le pourcentage utilisé de la capacité de stockage pour conserver des données effectives et non pas de données de redondance.

3.4.1.1 Niveau 1 : Disques dupliqués[1]

L'utilisation de disques dupliqués est une technique traditionnelle pour assurer la disponibilité des disques magnétiques. Il est évident que cette solution est la plus coûteuse car tous les disques sont dupliqués (G=1 et C=1), et à chaque écriture on doit aussi écrire sur le disque de redondance correspondant. Il y a alors une perte de 50% de la capacité de stockage.

[1] Mirrored Disks

Néanmoins, lorsque le nombre de contrôleurs est suffisant, la capacité de traitement d'opération de lecture par unité de temps est beaucoup plus importante que dans le cas d'un disque simple. Les performances globales de ces systèmes restent alors supérieures à celles de disques indépendants malgré la lenteur des opérations d'écriture.

Pour exploiter au mieux le parallélisme dans cette organisation, les fichiers sont entrelacés secteur par secteur (bloc par bloc) sur les N_G couples de disques, et puisque sur chaque groupe il n'y a qu'un seul disque de données, les transferts de grandes masses de données font intervenir en parallèle plusieurs groupes de disques.

3.4.1.2 Niveau 2 : Code de Hamming

Le problème de perte de l'information n'est pas propre aux disques magnétiques ; on l'avait déjà observé dans les mémoires DRAM. Des puces redondantes pour détecter et corriger les erreurs (code dit de Hamming) avaient été introduites à l'époque, ce qui augmentait la taille effective de la mémoire vive mais assurait la fiabilité de l'information.

Pour appliquer le même principe aux vecteurs de disques, l'information est entrelacée bit par bit (tout octet est distribué sur 8 disques), et des disques de vérification pour détecter et corriger les erreurs y sont ajoutés. Un seul disque de parité peut détecter une erreur, mais pour le corriger plusieurs sont nécessaires afin d'identifier le disque où l'erreur a eu lieu. Le code de Hamming impose que pour un groupe de 10 disques de données (G), il est nécessaire d'avoir 4 disques de vérification (C), pour G = 25, C=5 est indispensable.

Puisque l'unité de lecture/écriture des disques est toujours le secteur, un transfert d'au moins G secteurs est idéal pour ces systèmes. Tout transfert de taille inférieure nécessitera quand même la lecture des G secteurs pour assurer la fiabilité de l'information. Le niveau 2 est alors adéquat pour des systèmes où les E/S sont regroupées, et il n'est pas conseillé pour les systèmes de base de données où les unités de transfert sont souvent de petite taille.

3.4.1.3 Niveau 3 : Un seul disque de vérification par groupe

Le niveau 3 n'a pas pour objet d'améliorer le débit des transferts de petite taille ; il est plutôt destiné à réduire le prix du système, ce qui est réalisé avec la réduction du nombre de disques de vérification.

La plupart des disques de vérification du niveau 2 sont utilisés pour détecter le disque où l'erreur se situe, alors qu'un seul disque est nécessaire pour la détection de l'erreur proprement dite.

Comme on peut considérer que la majorité des contrôleurs d'aujourd'hui disposent de mécanismes capables de déceler les dysfonctionnements des disques (soit par des signaux d'interruption, soit par des informations de vérification stockées à la fin de chaque secteur), les disques de vérification utilisés pour la détection du disque en panne apparaissent alors inutiles pour cette tâche.

L'information endommagée peut être récupérée par comparaison entre la parité des disques en bon état et la parité calculée originellement (stockée dans le seul disque de vérification). Si les deux parités sont égales la valeur du bit du disque endommagée était 0, autrement elle était 1. Lorsque le disque endommagé est le disque de vérification, il faut simplement recalculer la parité des disques de données.

Le surcoût d'un système RAID de niveau 3 (C=1) se trouve alors entre 4% et 10% pour les valeurs de G considérées (1 disque de vérification pour 25 ou 10 disques de données). Les performances sont les mêmes que pour le niveau 2, mais la capacité de stockage effectivement utilisée est améliorée. Avec le niveau 3 le problème du surcoût dû à la redondance des informations est résolu ; il reste à améliorer le débit des accès aux données de petite taille. C'est ce à quoi les prochains niveaux RAID apportent des solutions.

3.4.1.4 Niveau 4 : Lectures/écritures indépendantes

La distribution des données sur tous les disques a l'avantage que toute la bande passante des E/S du vecteur est utilisée et donc que le temps de transfert est réduit. Cependant cette distribution limite le nombre d'opérations d'E/S qui peuvent être effectuée en parallèle à une seule, parce que justement tous les disques sont utilisés pour servir une requête. Les niveaux 2 et 3 n'exécutent en même temps qu'une opération d'E/S par groupe ; et à moins que les disques ne soient synchronisés, leur temps de positionnement et leur latence de rotation sont pénalisés à cause de la participation de plusieurs disques.

L'objectif du niveau 4 est d'améliorer le temps de transfert des petites quantités de données grâce au parallélisme. Les données ne sont plus réparties bit par bit, mais l'unité d'entrelacement est le bloc ; de cette façon plusieurs opérations d'E/S peuvent avoir lieu au même instant à l'intérieur d'un groupe. La parité est calculée bloc par bloc pour tous les disques de données du groupe et stockée sur le disque de vérification du groupe. Ainsi, le 8ème bloc du disque de vérification contient la parité du 8ème bloc de tous les disques de données.

Au niveau 2 le découpage de l'information, qui attribue un bit à chaque disque, est fait de telle sorte que le stockage des bits du code de Hamming soit fait de manière directe, simple et

efficace sur les disques de vérification. Déjà dans le niveau 3 le code de Hamming n'est plus utilisé, mais ce n'est que dans le niveau 4 que l'on profite de cette caractéristique pour regrouper l'information par blocs en y adjoignant des informations redondantes. De cette façon, une lecture peut être corrigée sans faire appel à d'autres disques. La différence entre les niveaux 3 et 4 est alors l'unité d'entrelacement de l'information : on passe du simple bit au bloc (de la taille du secteur).

Avec cette organisation une opération d'écriture concerne seulement deux disques : celui où l'information sera stockée et le disque de vérification. En effet, la nouvelle parité peut être calculée sans avoir à lire toutes les informations des autres disques mais uniquement en prenant en compte la parité enregistrée et les anciennes données stockées :

nouvelle parité = (ancienne donnée *xor* nouvelle donnée) *xor* ancienne parité

Une écriture de taille inférieure ou égale à un secteur ne nécessite donc que deux lectures et deux écritures sur disque alors qu'une lecture n'a besoin que de deux lectures. Malgré la forte réduction dans le nombre de disques accédés dans une opération, il faut noter que toute opération sur le vecteur utilise le disque de vérification, celui-ci devient donc rapidement un goulot d'étranglement pour le système.

3.4.1.5 Niveau 5 : Intégration des données et de l'information redondante

Malgré les modifications apportées par le niveau 4, chaque groupe dans le vecteur continue à n'accomplir qu'une seule requête de lecture/écriture par unité de temps. En effet comme toute opération utilise le disque de vérification, deux opérations ne peuvent avoir lieu au même instant. Pour éliminer cet inconvénient le niveau 5 distribue l'information redondante sur tous les disques ; on parle alors d'intégration des données d'information et de redondance.

L'impact de cette modification sur les performances est important car maintenant plusieurs écritures peuvent s'exécuter en parallèle. De cette manière, à ce niveau, les performances par disque s'approchent fortement de celles du premier niveau, tout en distribuant l'information de façon à être plus performant pour les grands transferts de données.

La méthode exacte utilisée pour la distribution de la parité affecte les performances d'un système RAID du niveau 5. Les travaux d'E.K. Lee et R.H. Katz [LK91] ont montré que la meilleure stratégie pour distribuer la parité, du point de vue des performances, consiste à attribuer la parité en faisant en parcours des disques, un bloc de chaque disque à la fois, et de stocker la parité des blocs dans le dernier disque avant de commencer un nouveau tour. Le disque utilisé pour stocker la parité sert ensuite de départ pour commencer le tour suivant. Cette méthode de

distribution diminue les conflits entre les disques lors du transfert de grandes quantités d'information. La figure 3.6 montre cette stratégie de placement de la parité ; ici P0 correspond à la parité des blocs 0, 1, 2 et 3, P1 à celle des blocs 4, 5, 6 et 7, et ainsi de suite.

Les concepteurs de systèmes de stockage RAID ne se sont pas arrêtés à ce niveau ; ils ont également proposé une architecture de vecteurs à deux dimensions, avec des disques de parité par colonne et par ligne[1]. Le but de cette organisation est de construire des systèmes RAID qui supportent la panne de deux disques à la fois [BBBM94, Ja93]. Cette architecture est évidement très fiable mais le surcoût engendré pour une écriture (six accès) n'est justifiable que pour seulement très peu d'applications.

0	1	2	3	**P0**
5	6	7	**P1**	4
10	11	**P2**	8	9
15	**P3**	12	13	14
P4	16	17	18	19

Figure 3.6 Distribution de la parité dans un système RAID de niveau 5.

3.4.2 Comparaison des niveaux RAID

Nous terminons cette présentation par la comparaison des caractéristiques qui sont déterminantes dans le choix d'un système de stockage. Cette tâche n'est pas simple car si on peut facilement se mettre d'accord sur les mesures de base à évaluer (fiabilité, performances et coût), il existe beaucoup de façons de les mesurer, et chacune dépendra finalement des besoins en matière d'E/S de chaque application. En général la méthode de comparaison est donnée par le but de celle-ci et par l'usage auquel est destiné le système global.

La figure 3.7 [PGK88] compare les trois paramètres qui déterminent le choix d'un système RAID pour un sous-système universel d'E/S à hautes performances : la capacité de stockage utilisable, l'amélioration des performances par disque pour les transferts de données de petite taille et l'amélioration des performances par disque pour les transferts de grande taille. Comme nous l'avons dit, la durée de vie de tous les niveaux RAID est amplement suffisante, c'est pourquoi nous n'y faisons pas référence lors de notre comparaison.

Par rapport aux contraintes que nous nous sommes fixés dans notre travail, contraintes qui sont celles des supports système valables pour tout type d'applications et qui puissent tirer profit du

[1] Dans [CLGK94] cette organisation est présentée comme le niveau 6 des organisations RAID.

parallélisme massif, il apparaît que des 5 niveaux RAID, seuls les niveaux 1 (disques dupliqués) et 5 (disques de données et de vérification confondus) répondent aux critères. En effet, les limitations des autres niveaux pour traiter les requêtes sur de petites quantités de données les excluent pour un usage général.

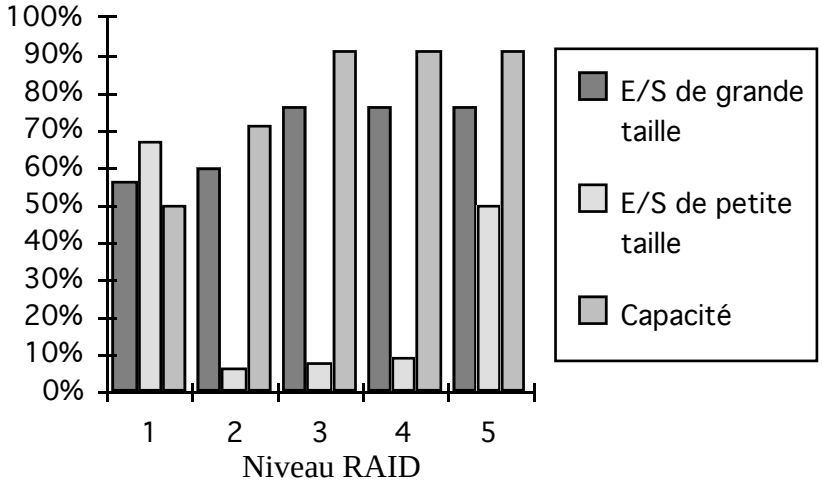


Figure 3.7 Comparaison des niveaux RAID.

Les résultats de la figure 3.7 ne sont pas étonnants si l'on analyse de près les différentes configurations RAID : les niveaux 2, 3 et 4 présentent tous un problème de congestion pour l'accès aux disques de vérification. En effet, dans ces trois niveaux chaque opération d'E/S, si petite qu'elle soit dans la taille de données à transférer, nécessite l'accès à tous les disques du groupe ce qui interdit l'exécution en parallèle de plusieurs requêtes d'E/S.

En revanche les niveaux 1 et 5 n'utilisent que deux disques par opération, ce qui favorise l'exécution en parallèle d'un grand nombre de requêtes, d'où leurs meilleures performances. Par ailleurs, si les niveaux 1 et 5 assurent tous deux un temps de réponse acceptable quel que soit le type de requête, le premier est en moyenne légèrement plus performant, mais au prix d'une grosse perte dans la capacité de stockage. Le choix de l'un ou de l'autre est donc économique, mais le niveau 5 reste plus réaliste pour être adopté par un système massivement parallèle.

3.5 Conclusion

Au cours de ce chapitre nous avons présente les unités de disques qui sont la base de tout système d'E/S parallèle. Nous avons tout d'abord mis en évidence les énormes différences d'évolution technologique des divers composantes d'un système informatique, qui font des unités de disques une barrière au développement des architectures à hautes performances. Cet

écart n'est pas près de disparaître et de nouvelles organisations pour ces unités sont indispensables pour atténuer ce retard.

Trois facteurs sont dominants dans les performances d'une unité disque : le temps de positionnement, la latence de rotation et le temps de transfert. Les solutions classiques pour améliorer un ou plusieurs de ces facteurs manquent de robustesse et leurs résultats leur donnent un champ d'application assez restreint. Les vecteurs de disques apparaissent alors comme la solution la plus adéquate, mais là encore la fiabilité du système est mise en question.

Les systèmes RAID comblent ces défaillances en proposant 5 organisations pour les E/S à hautes performances, qui s'appuient directement sur deux concepts orthogonaux : la distribution de l'information pour améliorer les performances et la redondance pour améliorer la fiabilité. Notre analyse de ces organisations nous a permis de ne retenir que deux de ces organisations, celles du premier (disques dupliqués) et du cinquième niveau, (intégration des données et de l'information redondante) car à la différence des autres elles ne favorisent pas un type d'accès aux informations aux dépens d'un autre. Nous avons montré que pour un système qui se veut général, les niveaux 1 et 5 offrent le meilleur compromis par rapport aux performances pour des requêtes de petite et grande taille, etc. En revanche, la considération de la capacité effective de stockage montre la supériorité du niveau 5 pour son intégration aux systèmes massivement parallèles. Cependant, la recherche dans le domaine des organisations des vecteurs de disques doit se poursuivre car en utilisation courante, c'est-à-dire, lorsqu'il s'agit de servir à la fois des transferts de gros volumes et de petits volumes d'information, les performances de ces systèmes se sont vues fortement dégrader [PGK88].

Ce chapitre achève la présentation de l'architecture matérielle des E/S. Le reste de ce rapport est consacré entièrement à l'organisation logicielle des sous-système d' E/S.

Chapitre 4
Les Systèmes Gestionnaires de Fichiers

Le système de fichiers (SF) ou système gestionnaire de fichiers (SGF)[1] est la composante logicielle principale d'un sous-système d'E/S. Ce chapitre est entièrement consacré à une présentation de l'état de l'art dans ce domaine pour les architectures parallèles et massivement parallèles.

Dans leur utilisation courante (sur des machines monoprocesseur), les systèmes d'exploitation monoprocesseurs sont souvent jugés par les utilisateurs en fonction de leur système de fichiers. En effet, les applications, dans leur grande majorité, font appel au SF pour la gestion de leurs données, d'où l'importance de l'efficacité et de la facilité d'utilisation du SF.

Le rôle principal des systèmes d'exploitation parallèles et massivement parallèles est de faciliter aux applications l'accès aux hautes performances offertes par ces architectures. Le SF est une composante très importante pour ces systèmes car pour beaucoup d'applications les performances globales sont fortement liées aux performances du sous-système d'E/S. Par ailleurs, le SF reste toujours une partie très considérable de la vision d'un système d'exploitation pour les utilisateurs, et donc de la simplicité d'utilisation de ces systèmes.

Dans ce chapitre nous étudions les projets en cours qui visent à doter les systèmes d'exploitation d'un SF adapté aux besoins des architectures parallèles. Nous commençons par une courte présentation des SF répartis afin de souligner les différences de fond entre les SF pour les systèmes parallèles et ceux proposés pour les systèmes distribués (réseaux locaux) ; ensuite, nous présentons les SF parallèles les plus représentatifs de la recherche dans ce domaine.

[1] Dans la littérature les deux termes sont utilisés indifféremment, bien qu'à l'origine l'un, SF, faisait référence à l'organisation des fichiers en disque (l'arborescence des répertoires en UNIX par exemple) ; et l'autre, SGF, concernait plutôt le programme système qui gère cette organisation.

4.1 Les systèmes de fichiers répar SFR)

Dans les systèmes d'exploitation centralisés, le SF est un programme tiel qui mémorise des informations relatives aux blocs libres des disques, aux blocs occup chaque fichier, au propriétaire de chaque fichier et aux droits d'accès des fichiers. Un fi st stocké dans son intégralité sur un seul disque et une structure de répertoires est géné t fournie. Le SF d'UNIX [Ba86] est l'exemple le plus répandu des SF centralisés.

Le modèle simple des systèmes centralisés où à une unité centra ue sont attachées quelques unités de disques, n'est plus valable dans les systèmes répart s ces systèmes on dispose de plusieurs machines qui communiquent via un réseau d'interc on, et de plusieurs disques connectés à une ou plusieurs de ces machines ; des stations de sans disques sont souvent présentes (voir figure 4.1). Un fichier est toujours stocké seul disque mais contrairement aux SF centralisés, les requêtes d'accès à un fichier ont en génér une machine autre que celle où le fichier est stocké.

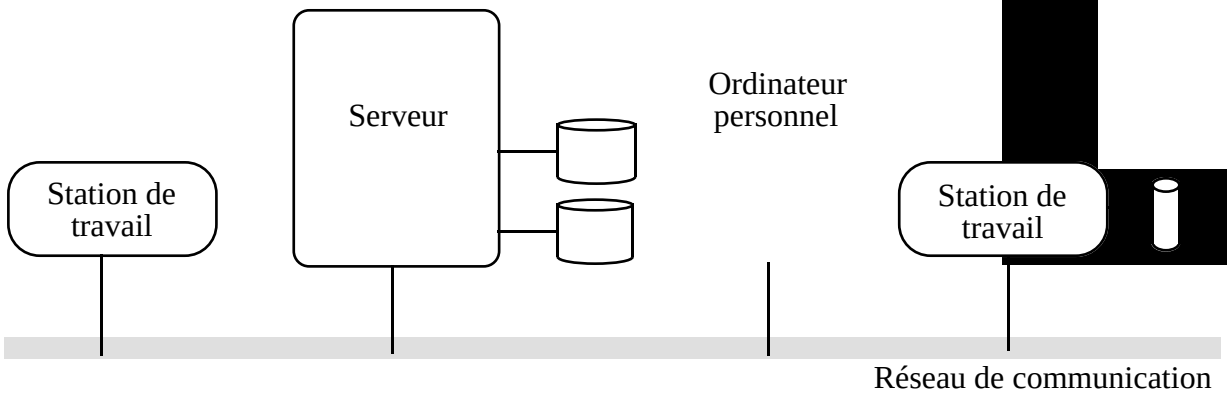


Fig. 4.1 Architecture d'un système réparti.

Idéalement l'utilisateur d'un SFR ne devrait pas se soucier de la localisation physique des fichiers ; le SFR devrait être capable de localiser un fichier demandé par un utilisateur et de le lui rendre accessible. Cette caractéristique de transparence de désignation est la grande différence entre un vrai SFR et un SF réseau comme NFS [Sa85][1], Swift [CL91] ou Zebra [HO93]. Dans un SF réseau, l'utilisateur n'ignore pas l'existence de plusieurs machines dans l'environnement et il est censé connaître la machine où se trouvent les fichiers qui l'intéressent. Dans le meilleur des cas, des arborescences de fichiers peuvent être installées localement, mais les mouvements

[1] NFS (Network File System) est d'ailleurs aujourd'hui le standard de facto des SFR, bien qu'il n'en soit pas un.

de fichiers entre machines est toujours visible. Un SFR relève l'utilisateur de cette tâche grâce à des systèmes de *désignation*[1] : l'accès aux fichiers est transparent vis-à-vis des machines [Ta90].

4.1.1 Désignation de fichiers

La résolution de noms pour les fichiers est un aspect très important de tout SFR. Le but d'un système de désignation est d'assurer l'unicité d'un nom de fichier, de façon à cacher tout ce qui concerne la distribution physique de l'environnement (si pour des raisons de performances par exemple un fichier change de localisation, son identification de référence ne doit pas changer). Lorsqu'un client spécifie ce nom dans une primitive d'E/S, le SFR doit être capable de localiser le fichier correspondant facilement.

Un SF comporte au moins deux niveaux pour les noms : interne et externe. Le *nom interne* (ou UID pour *"Unique IDentifier"*) est attribué et utilisé à l'intérieur du SF. Le *nom externe* (ou nom symbolique) est un mnémonique attribué par l'utilisateur. Quels que soient les noms accessibles aux utilisateurs, ceux-ci doivent persister aussi longtemps que les objets qu'ils désignent. C'est le service de désignation de fichiers qui assure la correspondance entre les noms externes et internes. Ce service peut, soit être intégré à la fonction de stockage, soit en être séparé [BBK91].

Lorsque la désignation est intégrée au stockage, comme dans le cas d'Unix, seuls les noms symboliques sont visibles des utilisateurs, les UID restant internes au SF et chaque serveur est responsable de ses propres noms. Ceci rend la traduction de noms symbolique en UIDs relativement chère car le nom symbolique est traduit morceau par morceau, et ceci nécessite de l'accès à un répertoire sur disque pour chacune des composantes du nom symbolique. En outre, seuls les types d'objets connus par le SF peuvent être nommés. Parmi les SFR les plus connus, nous citons Cedar [GNS88], AFS/Coda [Sa90] comme exemples de réalisations qui intègrent la désignation au stockage. Le projet Ficus de l'université UCLA [GHMP90] dispose aussi à sa base de ce type de mécanisme. Enfin, Chorus possède une désignation à moitié intégrée, avec possibilité de désignation d'objets inconnus par le SF [Pa88].

Certains systèmes ont donc dû séparer la fonction de stockage du *service de noms* pour assurer ce service indépendamment de la nature des entités nommées. Le service de noms n'interprète pas les UIDs, et on peut donc y référencer toute sorte d'entités : processus, fichiers, utilisateurs, machines, etc. Une application peut alors manipuler directement les UIDs et accéder ainsi aux

[1] Le terme nommage est aussi souvent utilisé comme traduction du mot anglais *naming.*

objets sans passer par un service de traduction à chaque nouvelle référence. Les systèmes d'exploitation Amœba [MT86] et SOS [NS89] utilisent une désignation de ce type.

La localisation d'un fichier à partir de son nom externe ou interne reste le problème majeur des SFR et comme nous l'exposons dans la suite, sa solution est directement liée au service de désignation.

Dans le cas de la désignation intégrée, la recherche d'un nom du fichier dans les répertoires effectue en même temps la localisation du fichier. Ceci est avantageux pour les performances de la localisation (en dépit de celles de la traduction du nom), mais a l'inconvénient de faire de la désignation une opération dépendante de la localisation : il n'y a pas de transparence de désignation.

Pour les SFR à désignation séparée, la recherche d'un fichier peut être faite à partir de son nom externe ou de son nom interne. La seule condition pour permettre la migration et/ou la réplication de fichiers individuels, est qu'il doit exister une totale indépendance entre le nom (externe ou interne) d'un fichier et sa localisation. Il existe une méthode simple pour résoudre le problème de localisation d'un SFR à désignation séparée, qui consiste à maintenir un tableau de localisation physique, indexé par les noms symboliques des fichiers (ou par l'UID) et à le rendre disponible à tout le réseau, soit à travers un serveur de noms, soit par sa réplication sur les diverses machines. Or, ces deux approches sont très pénalisantes dans un environnement extensible : d'une part le serveur de noms devient rapidement un goulot d'étranglement pour le système et rend le système réparti dépendant d'une seule machine ; d'autre part la réplication pose des problèmes de disponibilité et de mise à jour des différentes copies.

Dans [Ta90], l'auteur présente les deux approches qui semblent s'adapter le mieux à la localisation des entités à partir de leur nom symbolique dans les environnements distribués sous contraintes d'extensibilité : les tableaux de préfixes du système Sprite [WO86] et la désignation orientée utilisateur du système QuickSilver [CW87].

Un objectif important de tout SFR est d'offrir une grande disponibilité. La panne ou la déconnexion du réseau d'un disque ne doit pas entraîner la perte d'accès à un fichier particulier. La solution simple utilisée dans les systèmes centralisés, qui préconise la duplication de chaque disque n'est plus applicable à cause du grand nombre et de la diversité des unités qui peuvent coexister dans un environnement réparti.

4.1.2 Réplication

La technique utilisée pour assurer une bonne disponibilité des fichiers dans un SFR est la *réplication* de fichiers. Chaque fichier est alors copié sur plusieurs machines, ce qui augmente la probabilité de toujours retrouver une copie d'un fichier donné. La réplication de fichiers non modifiables (accès en lecture exclusivement) ne pose aucun problème. En revanche, si les mises à jour sont possibles, il se pose le problème crucial de la cohérence entre les différents exemplaires.

Le problème de cohérence étant très complexe, plusieurs types de systèmes de cohérence ont été proposés [Mo93], ce qui fait qu'aujourd'hui il existe une certaine souplesse dans la gestion de la cohérence. La politique la plus simple laisse à la charge de l'utilisateur la gestion de la cohérence de son information ; chaque processus dispose de sa propre copie, et n'est pas informé des mises à jour concurrentes. D'autre part dans une politique du type *copie maîtresse*, un processus peut lire de n'importe quel exemplaire du fichier, mais pour sa modification il faut toujours accéder en exclusion mutuelle à un exemplaire unique considéré comme la "copie maîtresse" du fichier. Toute mise à jour se fait sur cette copie, et est répercutée immédiatement[1] vers les autres exemplaires. Si la copie maîtresse est en panne, le fichier n'est disponible qu'en lecture, il ne peut alors plus être modifié.

La désignation et la réplication sont donc les deux axes principaux de la recherche dans le domaine des SFR. Cependant la sécurité de l'information dans les systèmes distribués, la gestion des différentes versions des fichiers (existence de caches locaux pour des fichiers distants) et la récupération des données après une panne (gestion d'un journal de bord) [Ta90] suscitent aussi de vifs intérêts.

Dans les systèmes répartis les gestionnaires de fichiers n'utilisent le parallélisme ni dans le SFR lui-même, ni dans les applications (à moins qu'il y ait des machines parallèles sur le réseau), ni dans les fichiers qu'elles manipulent [Co93]. Ce n'est que dans les systèmes de fichiers orientés parallélisme que ces aspects sont pris en compte.

4.2 Les systèmes de fichiers parallèles

L'inadaptation des SFR au parallélisme vient du fait qu'ils sont composés d'un certain nombre de programmes indépendants qui communiquent pour assurer la gestion de l'ensemble de l'information. La structure d'un SF parallèle est toute autre ; ici les entités qui le composent ne

[1] La modification et la mise à jour des copies forment une opération atomique.

sont plus de programmes indépendants, mais des processus qui forment un même programme parallèle et qui s'exécutent en parallèle sur des processeurs différents de la même machine.

Seule une architecture logicielle bien adaptée peut rendre un SF parallèle capable de fournir une puissance d'E/S en rapport avec la puissance de calcul des machines parallèles. Elle doit pour cela non seulement tirer profit des organisations matérielles étudiées dans les chapitres précédents, mais aussi intégrer le parallélisme à tous les niveaux du SF.

Le principe général consiste à exécuter les différents processus du SF sur des processeurs dédiés aux E/S et à fractionner les fichiers entre divers disques. Ainsi, une application (parallèle) peut accéder simultanément aux différentes parties d'un même fichier : des processus de la même application peuvent ainsi générer des requêtes d'E/S indépendantes (cf. Section 2.2) qui pourront être traitées en parallèle par les processus concernés du SF qui, eux, accéderont éventuellement à des disques distincts dans la machine.

Durant ces dernières années une série de projets ont démarré pour la conception de SF parallèles. Dans la suite nous présentons les trois réalisations de SF parallèles qui nous semblent les plus illustratrices de l'état de l'art : Bridge, un des projets pionniers dans la conception de SF parallèles développé à l'université de Rochester aux États-Unis, Hurricane de l'université de Toronto (Canada), et à titre d'exemple d'un produit commercial nous détaillons la structure de CFS des laboratoires Intel. Enfin nous passons en revue d'autres projets de recherche dans le domaine mais plutôt que de faire une description complète nous soulignons les différences dans leur approche par rapport aux SF détaillés.

4.2.1 Bridge

Bridge est un système de fichiers conçu comme un programme parallèle pour améliorer les performances des opérations d'E/S dans les machines multiprocesseurs [DS89, DSE88, Di90]. L'architecture cible est une machine parallèle BBN Butterfly à mémoire partagée du type NUMA. Bridge a été implémenté sur cette machine comme une simple application utilisateur au-dessus du système d'exploitation Chrysalis [Ch87] ; aucune modification du système d'exploitation n'a été nécessaire. Comme son but est seulement la validation des concepts, les disques ont été simulés en mémoire, et à chaque processeur on associe la gestion d'un disque virtuel. Bridge utilise les mécanismes de communication inter-processus offerts par Chrysalis : échange de messages implantés par des tampons dans la mémoire partagée.

Dans le prototype de Bridge, chaque fichier est distribué dans un système de stockage multiple. L'approche est basée sur une répartition physique des données, qui préserve la structure logique

des fichiers. Comme l'interface classique y est conservée, l'exécution des programmes déjà existants est toujours possible. Pour les programmes qui nécessitent de hautes performances d'E/S, une extension de cette interface est proposée.

La figure 4.2 présente les trois types de processus qui composent Bridge. Chaque processus peut s'exécuter sur un processeur différent et l'application de l'utilisateur peut être une application parallèle ou séquentielle :

- Le Serveur Bridge qui constitue l'interface entre le SF et les applications utilisateur,
- les programmes Utilitaires qui sont des applications intégrées au SF,
- les SF locaux qui correspondent aux SF conventionnels et qui opèrent indépendamment les uns des autres. Chaque processeur disposant d'une unité de stockage exécute un SF (nœud SF).

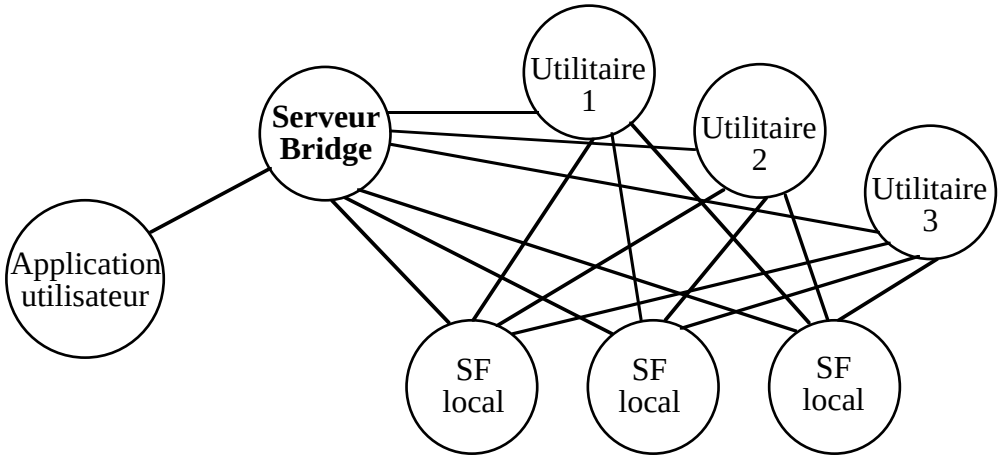


Figure 4.2 Architecture logicielle de Bridge.

La technique de répartition utilisée par Bridge est celle des fichiers entrelacés, où l'enregistrement logique est l'unité d'entrelacement, et où tous les SF locaux sont utilisés pour entrelacer chaque fichier. L'un des problèmes majeurs qui s'est posé est celui de la contiguïté de l'information : lorsqu'un bloc au milieu d'un fichier entrelacé est supprimé, faut-il compacter tout le fichier afin de garder l'accès direct aux blocs par des fonctions de hachage ? La solution retenue par Bridge consiste à chaîner les blocs dans une liste pour éviter tout compactage. Evidemment, le temps d'accès à un bloc quelconque d'un fichier est considérablement augmenté car l'utilisation d'une fonction de hachage pour retrouver le disque correspondant (cf. section 3.3.3) n'est plus possible.

Au niveau le plus bas, Bridge est composé des pilotes de périphériques qui gèrent les accès physiques aux données. Ces pilotes sont tout à fait conventionnels car chacun ne communique qu'avec un seul des SF locaux du système.

4.2.1.1 Le Serveur Bridge

Le serveur Bridge est l'interface entre le système de fichiers et les programmes de l'utilisateur. Sa fonction est de regrouper les différents SF locaux en une seule structure logique. Dans Bridge ceci est implanté comme un processus qui s'exécute sur l'un des processeurs contrôleur de disque ; cependant selon ses auteurs cela n'est pas une nécessité et une implantation distribuée est tout à fait envisageable.

Le serveur prend à sa charge la gestion des noms symboliques des fichiers. A chaque nom symbolique de fichier il associe un couple formé de l'identification du SF local chargé de la gestion du fichier (celui qui détient le premier bloc du fichier) et du nom du fichier à l'intérieur de ce SF.

Le serveur offre trois interfaces différentes du système selon les besoins de l'utilisateur : l'interface standard, l'interface parallèle et l'interface utilitaires.

Interface standard

La raison d'être de cette interface est la compatibilité avec les SF non parallèles. Elle propose des accès ordinaires pour les applications qui ne veulent pas se soucier de l'aspect entrelacé des fichiers. Les opérations offertes à ce niveau sont de type séquentiel : `open`, `read`, `write`, etc. Les requêtes des applications sont adressées au serveur qui les transmet aux SF appropriés.

Bien que le parallélisme ne soit pas explicite, le serveur Bridge est capable de l'exploiter. En effet, une requête d'E/S sur plusieurs enregistrements logiques (unités d'entrelacement) est divisée automatiquement en plusieurs sous-requêtes (une requête par enregistrement) envoyées simultanément aux SF locaux correspondants. Notons qu'un SF local peut alors recevoir plusieurs sous-requêtes, une pour chaque bloc à sa charge concernant la requête originale.

Interface parallèle

Il s'agit d'offrir des lectures et des écritures en parallèle de plusieurs blocs d'un même fichier. Pour ce faire, une nouvelle primitive est offerte : `parallel_open ()`. L'ouverture d'un fichier en parallèle regroupe plusieurs processus d'une même application dans un "*job*". Le processus qui exécute un `parallel_open` devient le propriétaire du job et spécifie au serveur Bridge l'identification d'une zone de transfert associée à chaque processus dans le job. Ensuite, les transferts entre les fichiers et ces zones sont effectués en parallèle.

La création d'un job de *t* participants a pour objectif le transfert en parallèle de *t* enregistrements logiques à chaque opération de lecture/écriture. Lorsque le propriétaire du job exécute une opération de lecture sur un fichier ouvert avec `parallel_open`, chacun des *t* participants du job reçoit en parallèle un enregistrement du fichier pour son traitement. Inversement, une écriture aura pour effet le transfert de *t* enregistrements (un par participant) vers les unités de stockage. C'est le SF qui exécute les transferts, sans que pour cela les processus du job soient prévenus. Aucune synchronisation n'est faite par le SF car Bridge considère que le SF ne dispose pas d'informations suffisantes pou réaliser cette synchronisation judicieusement. C'est alors au propriétaire du job de synchroniser tous ses processus.

L'interface parallèle masque la structure parallèle des fichiers. L'application ne connaît même pas le degré de parallélisme offert par le SF. Si un job est composé de plus de processus (*t*) que des SF locaux (*p*), Bridge effectue autant d'accès, de *p* enregistrements chacun, que nécessaires pour l'exécution complète d'une requête (transfert de *t* enregistrements). Il virtualise ainsi le parallélisme, et le niveau d'entrelacement des fichiers reste transparent aux applications.

Interface utilitaires

Pour profiter au maximum de l'entrelacement des fichiers, une interface de bas niveau qui s'adresse aux développeurs système, pour le développement des utilitaires, est également offerte. Les utilitaires Bridge sont des applications spécialisées (`copy, sort, grep,` etc.) qui s'intègrent au système de fichiers, et contrairement aux applications des utilisateurs, ils ont une connaissance particulière de la structure d'un fichier et des SF locaux.

L'interface pour les utilitaires offre une opération pour obtenir du serveur Bridge la structure interne d'un fichier. Une fois l'utilitaire connaît la distribution d'un fichier sur les différents SF locaux, l'interface lui permet d'accéder directement aux SF locaux , le serveur Bridge n'est donc plus nécessaire pour arbitrer toutes les opérations. En général un utilitaire se comporte en première instance comme client du serveur Bridge, mais ensuite il se substitue au serveur vis-à-vis des SF locaux.

4.2.1.2 Les utilitaires Brigde

L'interface parallèle exploite le parallélisme des fichiers mais génère des communications inter-processeurs inutiles pour un certain nombre d'applications. Par exemple, une application qui compte le nombre d'occurrences d'une chaîne de caractères dans un fichier nécessite le transfert de tout le fichier vers les processeurs de l'application, alors que le résultat attendu est seulement un entier. Pour répondre à cela, Bridge propose de définir un ensemble d'applications, les

utilitaires, qui s'exécutent directement sur les processeurs qui contrôlent les disques. Un utilitaire parallèle peut ainsi compter le nombre d'occurrence de la chaîne dans le fichier et transférer seulement le résultat vers l'application utilisateur.

4.2.1.3 Les systèmes de fichiers locaux

Les systèmes de fichiers locaux de Bridge sont basés sur les Systèmes de Fichiers Elémentaires (EFS pour *Elementary File System*) [SC84] développés dans le cadre du système distribué Cronus [GDS86]. Il s'agit de systèmes de fichiers sans état, avec un espace de désignation plat (sans arborescence), et sans contrôle d'accès (celui-ci est offert par le serveur Bridge). Chaque EFS est un SF conventionnel qui se suffit à lui même et opère indépendamment des autres. Les fichiers sont représentés comme une liste doublement chaînée de blocs ; un pointeur sur le premier bloc du fichier est placé dans une entrée du répertoire du SF. EFS utilise un cache en mémoire pour les blocs les plus récemment accédés.

Le choix d'EFS comme serveur local de Bridge obéit à des raisons pragmatiques et technologiques. D'une part EFS existait déjà pour la machine cible, et son code source était disponible ; d'autre part l'absence de mécanismes de protection et de gestion d'état permet aux concepteurs de Bridge d'implémenter, sans redondance, ces fonctionnalités au niveau du serveur.

4.2.2 Le SF du système Hurricane

Hurricane [SUK92] est un système d'exploitation pour machines MIMD à mémoire partagée du type NUMA, dont le nombre de processeurs peut être très important. Il est conçu comme un micro-noyau et est basé sur le concept de "*cluster*"[1] qui est l'unité de partage du système. Un cluster représente en soi une machine parallèle complète (processeurs, mémoires, disques, etc.) et offre toutes les fonctionnalités d'une architecture fortement couplée (mémoire partagée). Un système parallèle avec un grand nombre de processeurs est donc ici décomposé en plusieurs clusters, de telle façon que chaque cluster gère un groupe unique de processeurs dits *voisins*. La notion de voisinage est relative au coût d'accès à la mémoire : les accès mémoire sont plus rapides à l'intérieur du cluster qu'à l'extérieur. Les clusters coopèrent entre eux par échange de messages pour offrir aux applications une vue intégrée du système.

[1] A ne pas confondre avec les clusters de ParX.

La conception du système de fichier d'Hurricane s'insère dans un contexte où l'extensibilité est un critère crucial : le SF d'Hurricane est conçu pour la gestion d'un système où la demande d'E/S augmente proportionnellement au nombre de processeurs de la machine. Pour ce faire, la réalisation du SF part de l'hypothèse que l'extensibilité est respectée au niveau matériel, c'est à dire que le rapport entre le nombre de processeurs et le nombre de disques de l'architecture est à peu près constant [KS93].

4.2.2.1 Approche architecturale

Le SF d'Hurricane est fortement influencé par les choix conceptuels adoptés pour le système d'exploitation. En premier lieu, il y a l'approche micro-noyau suivie ici, qui fait du SF un serveur implanté au-dessus de ce noyau. En effet, comme dans ParX, seuls les mécanismes de base pour la communication inter-processus et la gestion mémoire sont intégrés au noyau ; la plupart de services système sont en fait assurés par des serveurs qui s'exécutent au niveau utilisateur.

Deuxièmement, Hurricane est structuré en fonction du système de clusters ; le SF suit alors ce schéma et à chaque cluster associe un SF complet. Pour minimiser les communications inter cluster, les clusters disposent chacun d'un cache dans lequel est stocké l'état global du SF. Les SF locaux ne communiquent alors entre eux que pour maintenir cet état à jour et pour distribuer les requêtes d'E/S à travers les différents clusters.

Troisièmement, Hurricane est un système d'exploitation à un seul niveau de stockage, tel qu'il l'avait été introduit dans Multics [Or72] : la mémoire vive est considérée comme un cache de la mémoire secondaire. Ici, un fichier est attaché à une région dans l'espace d'adressage virtuel d'un processus et toute référence à une position mémoire dans la région est par conséquent une référence à la position correspondante dans le fichier. Ceci nécessite que le SF identifie tous les accès mémoire qui font référence à un fichier, plutôt que de recevoir des appels système explicites. En outre, dans un tel système il est naturel de laisser à la charge du gestionnaire mémoire la gestion des caches, tâche qui normalement revient au SF pour tout ce qui concerne les fichiers. Les interactions entre le SF et le gestionnaire mémoire doivent donc être considérées.

La dernière caractéristique d'Hurricane qui influe directement sur la conception du SF est l'appel local de serveurs (LSI pour *Local Server Invocation*). Cette fonctionnalité permet, d'une façon performante, d'invoquer le code et d'accéder aux données de serveurs entre différents clusters. La création de nouveaux processus dans l'espace d'adressage du serveur est également possible sur demande. Grâce à ce mécanisme l'état du SF peut être accédé par plusieurs

applications en parallèle dans les processus serveurs du SF. L'architecture du SF est ainsi simplifiée, car LSI permet la création de processus de travail lors de l'appel à un serveur, ce qui évite tout risque d'interblocage pour manque de processus au niveau d'un serveur.

4.2.2.2 Architecture des SF locaux (par cluster)

La structure du SF d'Hurricane est schématisée par la figure 4.3. Elle comprend trois serveurs au niveau utilisateur : le serveur de noms (NS pour **N**ame **S**erver), le serveur d'ouverture de fichiers (OFS pour **O**pen **F**ile **S**erver) de fichiers et le serveur de blocs de fichiers (BFS pour **B**lock **F**ile **S**erver). La figure montre les interactions possibles entre les différents serveurs. L'exécution d'une application nécessite l'allocation d'un cluster (cluster 1) ou de plusieurs pour une grande application parallèle (cluster 1 et 2). Les lignes grises représentent les communications additionnelles pour le traitement des défauts de page.

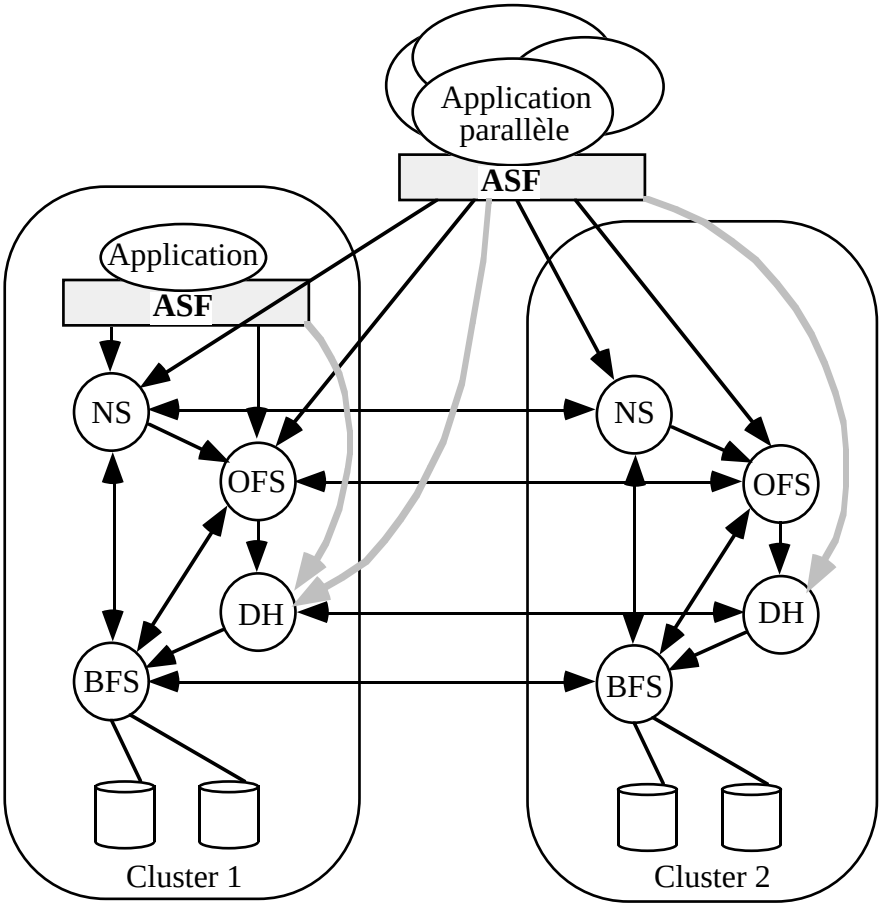


Fig. 4.3 Structure du SF d'Hurricane.

Le serveur de noms gère l'espace des noms et vérifie les requêtes d'ouverture de fichiers. Le serveur d'ouverture de fichiers maintient l'état de tous les fichiers ouverts et vérifie les requêtes

d'installation d'un fichier dans l'espace d'adressage de l'utilisateur. Le serveur de blocs contrôle le système de disques et détermine l'unité à laquelle une requête est destinée.

Il existe un serveur au niveau système, **D**irty **H**arry *(DH)*, qui prend en charge la mise à jour sur disque des pages mémoire modifiées. Lorsque DH retrouve dans la mémoire une page qui a été modifiée, il la transfère au serveur de blocs pour son écriture sur disque.

Le serveur ASF (**A**lloc **S**tream **F**acility) est une bibliothèque au niveau utilisateur qui traduit les appels du type `read` et `write` en accès aux régions mémoire où se trouvent les données des fichiers concernés. ASF supporte l'interface standard d'UNIX (stdio) et une extension propre qui permet de tirer profit de l'accès aux fichiers à travers des adresses mémoire [KSU92, KSU93].

Pour l'ouverture d'un fichier l'interaction entre les différents serveurs se déroule de la façon suivante : lors de l'appel à la primitive `open`, un message est délivré au serveur de noms par le serveur ASF ; le serveur de noms transforme la chaîne de caractères du nom en un jeton (un numéro unique qui identifie le fichier au système), vérifie les droits d'accès de l'application et transmet le message au serveur BFS ; puis BFS utilise ce jeton pour retrouver les informations associées au fichier et envoie le tout au serveur OFS ; enfin le serveur OFS rend à l'application la clé (capacité) d'accès au fichier.

Pour les appels du type `read` et `write`, la bibliothèque ASF envoie au serveur OFS une demande d'installation du fichier dans une région mémoire de l'espace d'adressage de l'application. Une fois vérifiée la validité de l'accès, OFS envoie le jeton du fichier (obtenu à partir de la capacité) au serveur mémoire. Si le fichier est déjà disponible en mémoire, ASF copie les données de ou vers la région mémoire correspondante.

Pour le premier accès à un fichier une faute de page est générée ; et ce serveur mémoire regarde d'abord dans le cache de fichiers si le fichier est disponible, dans le cas contraire, il envoie un message au BFS pour effectuer la lecture du fichier sur disque. BFS détermine l'unité disque qui contient le bloc recherché (ceci peut nécessiter le concours d'un BFS d'un autre cluster) et lance l'opération de lecture sur le dispositif. Pour le cas d'une écriture, c'est le serveur DH qui communique avec le BFS pour la mise à jour sur disque des blocs modifiés du fichier.

Afin d'intégrer tous les SF locaux dans une seule entité globale une approche orientée objet a été choisie. Selon cette approche les concepteurs d'Hurricane ont défini différents niveaux de fichiers, pour associer à chaque niveau une organisation des données particulière et une politique de cohérence spécifique. Ceci permet aux utilisateurs de choisir l'organisation physique et la politique de cohérence qui s'adaptent le mieux à leurs données.

4.2.2.3 Le serveur ASF

Le serveur ASF d'Hurricane est un service qui se base directement sur la mémoire partagée de l'architecture cible. D'une conception orientée objet, ASF offre au SF d'Hurricane trois avantages déterminants par rapport aux SF conventionnels : une amélioration importante des performances, une gestion correcte de la concurrence et le support pour une grande variété d'interfaces.

Performances

Par l'intermédiaire de son interface ASI (pour **A**pplication-level **S**tream **I**/O) et à l'aide du gestionnaire de mémoire virtuelle, ASF donne à l'application un accès direct aux tampons du système, plutôt que de faire une double copie mémoire des données : la première fois du tampon système vers le tampon de la bibliothèque d'E/S, et une seconde fois du tampon de la bibliothèque vers l'espace d'adressage de l'application. Evidemment, cette amélioration n'est possible que sur les architectures à mémoire partagée ; dans une architecture sans mémoire commune la copie mémoire entre le processeur d'E/S et le processeur où s'exécute l'application est inévitable.

Concurrence

ASF a été conçu pour servir des applications multi-threads qui s'exécutent sur des machines parallèles. ASF offre de primitives atomiques pour l'installation en mémoire des fichiers, ce qui permet de limiter le temps de verrouillage d'un fichier pour la copie de ses données : un fichier n'est pas disponible seulement pendant le temps que les structures de données internes de la bibliothèque sont modifiées.

En outre, toute opération d'E/S de l'interface ASI retourne un code d'erreur directement au thread appelant ce qui permet d'identifier clairement le thread qui a généré l'erreur (contrairement à la bibliothèque standard d'UNIX qui ne dispose que d'une variable d'erreur, `errno`, pour informer qu'une erreur a eu lieu).

Fonctionnalités

Grâce à l'indépendance entre les modules système et les modules interface, ces derniers peuvent offrir en même temps une grande variété d'interfaces aux applications (stdio d'Unix, E/S avec format de C, ASI etc.). Une application peut même mélanger dans son code des primitives d'E/S des différentes interfaces ce qui favorise le portage et la réutilisation du code. De plus,

une telle fonctionnalité facilite l'utilisation de l'interface ASI pour améliorer les performances des applications qui font un usage intensif des E/S.

4.2.3 CFS (Concurrent File System)

CFS est un système de fichiers parallèle des laboratoires Intel conçu pour les iPSC/2 et iPSC/860 [Pi89, AS89]. L'architecture de la machine cible et de son sous-système d'E/S sont illustrés par la figure 4.4. Les unités de calcul (UC) sont interconnectées selon une topologie en hypercube et ne partagent pas de mémoire. Les processeurs d'E/S (PES) sont attachés à certains nœuds de la machine, et chacun possède un contrôleur SCSI qui peut gérer jusqu'à sept dispositifs de stockage. Il n'y a pas de connexion physique directe entre les PES.

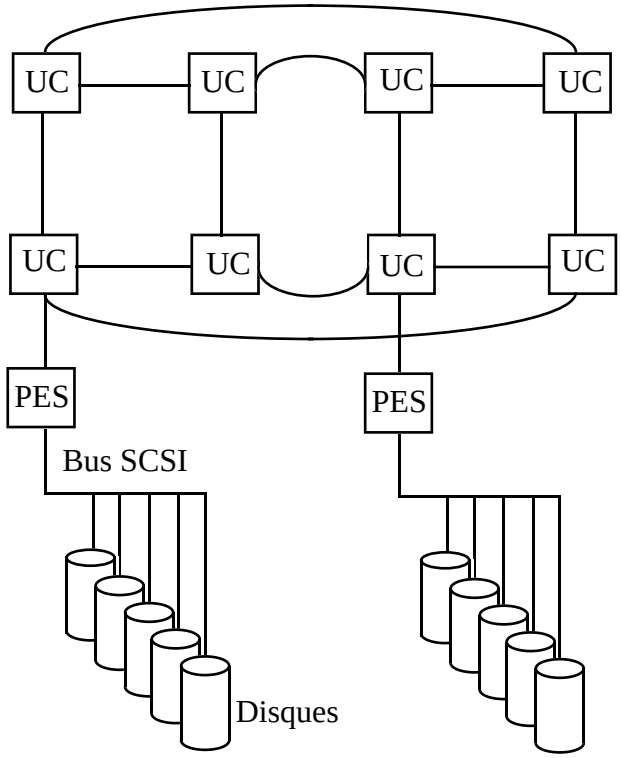


Fig. 4.4 Architecture de l'iPSC/2.

4.2.3.1 Architecture de CFS

CFS est un serveur de fichiers conçu pour s'exécuter avec le système d'exploitation NX/2. NX/2 est disponible aussi bien sur les UC que sur les PES et offre les services de transmission de messages inter-processus (avec routage) et de gestion de processus. Un PES n'exécute pas directement les applications utilisateur mais il fournit les services disque pour tous les utilisateurs.

CFS est composé de trois éléments principaux : les gestionnaires de disques qui s'exécutent sur chaque PES, un serveur de noms centralisé sur un des PES, et une bibliothèque de fonctions utilisateur qui est compilée avec les applications utilisateurs.

Gestion de disques

Chaque nœud d'E/S (PES plus disques) possède un SF complet qui est lié aux autres SF de manière à ce qu'un utilisateur puisse accéder à tout fichier dans le système. CFS offre une vue globale de tous les disques du système comme s'ils n'étaient qu'un seul, avec un seul et unique SF.

Afin de maximiser la concurrence dans l'accès aux fichiers, ceux-ci sont entrelacés non seulement sur tous les disques d'un contrôleur mais sur tous les disques du système. La philosophie de CFS est que cette distribution doit être homogène et non pas influencée par les besoins spécifiques d'une application. La distribution des fichiers est faite avec la politique du tourniquet par blocs de taille fixe[1]. Ceci se traduit par un entrelacement par groupes de blocs consécutifs au niveau des SF des nœuds d'E/S (autant de blocs consécutifs que de disques dans le nœud).

Pour accéder aux fichiers, les processus d'une application, qui s'exécutent exclusivement sur les UC, envoient leurs requêtes aux processus qui administrent les SF sur chaque nœud d'E/S, dits *processus disque*. Ces processus ont en charge le contrôle des disques du nœud, la gestion d'un grand cache et l'administration de la structure du SF local auquel il est associé ; il y a un unique processus par SF local, c'est-à-dire, que chaque PES exécute un processus disque.

A la création d'un fichier, un processus disque désigné par le serveur de noms devient le gestionnaire du fichier. Le rôle des gestionnaires de fichiers est de maintenir à jour un bloc d'en-tête pour chacun des fichiers dont il est responsable, et d'allouer des blocs disque à ces fichiers (grâce à une carte de blocs libres sous forme de tableau de bits). Parmi les informations contenues dans un en-tête on retiendra l'utilisateur propriétaire, la taille du fichier et surtout la carte des blocs du fichier où chaque bloc est associé à un couple disque/bloc. Pour de gros fichiers CFS utilise des blocs indirects, sans limitation de profondeur d'indirections. Le bloc d'en-tête est stocké dans le SF local au processus gestionnaire.

Pour accéder à un fichier les applications envoient leurs requêtes au processus disque gestionnaire du fichier, c'est-à-dire au PES auquel est connecté le disque qui contient le bloc d'en-tête du fichier. Pour la gestion des requêtes d'E/S, les processus disque utilisent une file

[1] L'implémentation actuelle utilise une taille de 4K, car cette taille maximise les performances des transferts de données via les liens de l'hypercube.

d'attente où peuvent se retrouver en même temps des requêtes de transfert de blocs concernant différents fichiers, en provenance des différentes applications, quels que soient le processus ou l'UC, et des requêtes d'allocation/libération de blocs en provenance d'autres processus disque.

Pour améliorer les performances CFS utilise un cache des blocs les plus récemment accédés pour chaque nœud d'E/S. La gestion de ces caches est effectuée par les processus disque et outre la gestion de cohérence, elle comprend un système de chargement anticipé de blocs et un système d'écriture différée. Ainsi, un accès aux disques comporte en général le transfert d'un minimum de 8 blocs.

Il existe deux autres processus, chacun sur un PES spécifique, qui ont en charge d'autres fonctions d'administration du système CFS. La tâche de ces processus est de garder la trace des applications en exécution ; ils informent les processus disque de la terminaison des applications afin de permettre à ceux-ci de fermer les fichiers correspondants. Ils interviennent également dans l'implantation de pipe-lines compatibles avec ceux d'Unix et dans le démarrage de CFS.

Désignation de fichiers (serveur de noms)

Les fichiers dans CFS sont identifiés par un nom d'accès unique (nom symbolique) : "`/cfs/...`" identifie un fichier CFS et le différencie des fichiers du SF de la machine hôte (frontal d'accès à l'hypercube). Le même appel d'E/S peut alors faire référence à un fichier CFS comme à un fichier de la machine hôte ; c'est le serveur de noms, qui à partir du nom du fichier, sélectionne le SF auquel adresser la requête. Le serveur de noms associe à chaque nom symbolique un numéro de bloc d'en-tête composé d'un numéro de volume (disque) et d'un numéro de bloc physique sur ce volume.

Une structure unique de répertoire compatible avec celle d'Unix est utilisée et gérée par un seul *serveur de noms* qui s'exécute sur un PES quelconque. Tous les fichiers de type répertoire du système sont administrés par ce processus : la hiérarchie est remplacée par un *fichier répertoire.*

L'unicité du processus de noms pour tout le SF est un choix qui évite tout problème de cohérence, mais qui peut générer un goulot d'étranglement pour les environnements où les opérations d'ouverture et de fermeture sont très fréquentes. Bien que l'hypothèse de départ selon laquelle les applications exécutent beaucoup plus d'opérations de lecture/écriture que d'opérations d'ouverture/fermeture de fichiers ne soit pas fausse, dans la pratique on s'aperçoit que cette décision pénalise fortement les performances de CFS [PFDJ89]. Ses concepteurs prévoient d'ailleurs de reconsidérer ce choix de centralisation [FPD93].

La bibliothèque CFS

Les appels d'E/S sont réalisés à travers une bibliothèque qui est liée aux applications. La bibliothèque transforme chaque requête d'E/S (appel système) en envoie de message au serveur de noms ou aux processus disque.

Lors de l'ouverture d'un fichier, la routine associée envoie une requête au processus serveur de noms pour vérifier les droits d'accès de l'application et obtenir l'identification du processus disque propriétaire du fichier. A partir de cette identification l'application récupère l'en-tête du fichier, ainsi elle sait à quel processus disque s'adresser pour chaque bloc. Un appel d'E/S est donc divisé en autant de requêtes qu'il y a de blocs, et chaque requête est envoyée au processus disque approprié.

Pour éviter tout problème de cohérence de l'information les UC ne gèrent pas de cache des blocs précédemment accédés. Comme nous l'avons vu, cette gestion est laissée à la charge des processus gestionnaires de disque, ce qui augmente les communications entre les UC et les PES. Cependant un cache en lecture exclusive des en-têtes de fichiers est conservé sur chaque UC pour éviter le transfert de ce bloc à chaque opération. Une convention simple entre les UC et les processus disque permet la mise à jour de ces caches.

4.2.3.2 L'interface de CFS

Les caractéristiques spécifiques de CFS peuvent rester transparentes à l'utilisateur : une application peut toujours utiliser l'interface standard d'Unix. Mais afin d'exploiter la puissance de CFS, Intel a développé un nouvel ensemble d'appels système qui permet l'accès aux fichiers CFS en parallèle, et compatible avec l'interface classique.

Deux appels spécifiques CFS sont offerts : `lsize` qui alloue entièrement l'espace nécessaire pour la création d'un fichier (si la taille finale du fichier est connue), ceci permet de pré-allouer tous les blocs du fichier en disque afin d'augmenter les performances à l'écriture ; et `restrictvol` qui permet à l'application de sélectionner le disque (ou les disques) où les données d'un fichier doivent être stockées (par défaut les fichiers utilisent tous les disques disponibles).

Par ailleurs, l'interface classique n'offre pas un contrôle des accès concurrents aux fichiers. Lorsque plusieurs processus accèdent en écriture au même fichier, le résultat ne peut être prévu ; ici, c'est le processus qui effectue la dernière écriture qui détermine l'état final du fichier. En outre, si un processus écrit dans un fichier et s'il en averti d'autres afin qu'ils puissent lire son

contenu, le résultat des lectures ne contiendra pas forcement les modifications effectuées, car les nouvelles données peuvent être encore en transit vers le disque.

Pour résoudre ces problèmes, de nouvelles opérations sont offertes au programmeur [AS89] : les primitives asynchrones `iread` et `iwrite`. A chacune de ces opérations un identificateur est retourné, ce qui permet aux applications de demander ultérieurement au système l'état des requêtes (fonctions `iodone` et `iowait`).

Quatre modes d'accès sont définis pour un fichier par l'appel à la primitive `setiomode` :

mode 0 : accès sans contrôle de concurrence. Tous les processus obtiendront des pointeurs indépendants pour accéder au fichier. C'est le mode d'accès attribué par défaut à chaque nouveau fichier.

mode 1 : accès séquentiel aléatoire. Un pointeur commun est géré et toute opération de lecture/écriture démarre à la fin de l'opération précédente (indépendamment du processus qui l'exécute).

mode 2 et 3 : lecture ou écriture coordonnée. Le segment du fichier à lire ou à modifier est calculé en fonction du numéro d'identification du nœud où la requête est générée. Le mode 2 utilise des segments de taille fixe, tandis que dans le mode 3 les tailles peuvent varier et un ordonnancement des opérations sur le disque est alors nécessaire.

4.2.4 Bilan des principaux SF parallèles

De cette courte présentation des SF il ressort deux caractéristiques fondamentales qui marquent la différence entre les SF parallèles : l'architecture du SF et l'interface proposée aux utilisateurs.

4.2.4.1 Architecture du SF

Bridge et CFS étant les premiers SF parallèles disposent d'une architecture simple composée d'un serveur centralisé et d'un ensemble de SF locaux qui résolvent en dernière instance les requêtes d'E/S. Une différence à noter entre ces deux SF, alors que dans Bridge les applications des utilisateurs ne communiquent qu'avec le serveur Bridge, dans CFS les applications s'adressent au serveur seulement pour l'ouverture des fichiers, ensuite elles communiquent directement avec les SF locaux.

L'existence d'un serveur centralisé dans ces systèmes pose un problème à leur extensibilité. En effet ces systèmes s'adressent à des architectures MIMD à mémoire distribuée avec un nombre de nœuds restreint. De plus, les évaluations des performances de ces systèmes ont montré que ce serveur devient rapidement un goulot d'étranglement du système. Bridge propose l'exécution

des certains utilitaires sur les nœuds mêmes où se trouvent les SF locaux, mais là encore, les utilitaires doivent s'adresser au serveur pour l'ouverture des fichiers.

Le SF Hurricane utilise une structure similaire où les fonctionnalités du serveur et des SF locaux sont réparties sur trois processus du SF et sur le gestionnaire de mémoire du système d'exploitation. Plutôt que d'avoir un serveur centralisé pour la désignation des fichiers, Hurricane propose de répliquer ce serveur sur chaque cluster de la machine. Les auteurs ne spécifient pas néanmoins comment ils garantissent la cohérence de ces serveurs.

La conception du SF est basée sur le gestionnaire de mémoire et sur le concept de cluster. En effet, un fichier est projeté sur la mémoire virtuelle et le SF traduit les accès aux fichiers en des opérations sur l'espace virtuel. Bien qu'une application puisse s'exécuter sur plusieurs clusters le système est conçu pour que chaque cluster représente le domaine de communication d'une application. La distribution d'un fichier sur des disques dans des clusters différents réduit fortement les performances du système.

4.2.4.2 Interface

Bridge et CFS proposent chacun une extension de l'interface classique qui permet aux applications de mieux exploiter le parallélisme offert par les sous-systèmes d'E/S. L'interface parallèle de Bridge se limite à une opération de `parallel_open` pour assurer le transfert en parallèle de plusieurs blocs d'un fichier vers un ensemble de processus spécifiés lors de l'ouverture du fichier. L'interface de CFS est plus souple et propose quatre modes d'accès concurrents aux fichiers, dont un avec un pointeur partagé par tous les processus d'une application parallèle.

L'interface ASI du SF de Hurricane bénéficie directement de la mémoire partagée de l'architecture cible du système. Le serveur ASF contrôle l'accès direct des applications aux tampons du système, ce qui limite le nombre de copies mémoire nécessaires lors d'une opération d'E/S. ASI est une interface pour exploiter la projection des fichiers sur l'espace virtuel dont l'objectif est d'offrir les mêmes fonctionnalités d'une interface classique pour des applications parallèles. Cependant aucune extension de l'interface classique n'est proposée, et de ce fait les applications ne peuvent exprimer aucun parallélisme dans leurs opérations d'E/S.

Par ailleurs, des trois SF étudiés, seul CFS offre à l'utilisateur une opération pour spécifier les unités de disques qui doivent être utilisés pour l'entrelacement d'un fichier. En revanche, aucun des trois SF ne permet la définition de la taille des blocs à entrelacer.

4.2.5 Autres SF parallèles

Les systèmes précédemment décrits ne sont pas les évidemment les seuls ; durant les dernières années la recherche dans les SF répartis et parallèles a connu un essor considérable. Plusieurs projets proposent aujourd'hui de nouvelles approches qui cherchent à atténuer la différence entre la puissance de calcul et le traitement des opérations d'E/S. Dans cette section nous discutons ces approches en faisant une liste des projets qui les introduisent et qui par conséquent servent de base au développement de notre travail.

4.2.5.1 Vesta [CBF93]

Vesta est un SF développé par le centre de recherche T.J. Watson d'IBM pour leur machine massivement parallèle Vulcan. L'objectif de Vesta est de faire face aux problèmes d'E/S des applications scientifiques numériques dans les architectures massivement parallèles. Vesta est fortement influencé par l'architecture de Vulcan : MIMD à mémoire distribuée basée sur des processeurs i860. L'allocation des processeurs aux utilisateurs est faite par division progressive de la machine, et tous les processeurs d'E/S (avec des dispositifs de stockage) forment une partition partagée pour tous les utilisateurs.

Vesta a été conçu pour fournir un service rapide des E/S dans les machines massivement parallèles. Vesta n'est pas prévu comme un système de stockage, mais plutôt comme un niveau intermédiaire entre la machine et un système de stockage massif qui n'est pas directement lié à Vulcan. Les fichiers sont chargés dans le système Vesta en même temps que l'application qui va les utiliser. Les données sont donc supposées distribuées de façon optimale sur l'ensemble de disques de Vesta pour chaque exécution d'une application. Les auteurs ne prennent pas en compte le temps de ce chargement initial dans leur évaluation des performances.

La structure logicielle de Vesta ne nécessite pas d'un serveur global intermédiaire entre les applications des utilisateurs et les serveurs de fichiers sur chaque nœud d'E/S. En effet, comme la distribution des fichiers est calculée à chaque lancement de l'application qui les utilise, une fonction de hachage est suffisante pour déterminer à partir du nom symbolique du fichier, l'unité de disque sur laquelle se trouve l'en-tête du fichier.

L'interface de Vesta offre aux utilisateurs trois types d'opérations sur un fichier : des opérations sur le fichier comme entité du système, p. ex. `create`, `delete` ; des opérations pour accéder à un fichier, p. ex. `open` et des opérations d'accès aux données d'un fichier, p. ex. `read`, `write`. Il existe aussi des primitives qui permettent le transfert d'un fichier entre Vesta et un SF externe, et des opérations pour placer des points de reprise sur les modifications apportées à un fichier, de

manière à ce qu'on puisse toujours revenir en arrière jusqu'à un point stable dans l'exécution d'une application.

L'accès à un fichier est fait en deux étapes. Premièrement, un processus de l'application qui veut accéder au fichier exécute une opération `attach` qui prévient Vesta de l'intention de l'application d'accéder le fichier. Vesta vérifie les droits d'accès au fichier, diffuse l'autorisation à tous les nœuds d'E/S qui contiennent des données du fichier, et renvoie au processus appelant les paramètres du fichier. L'opération `attach_share` permet de partager de cette information entre plusieurs processus. Puis, chaque processus exécute un `open` qui détermine la partition du fichier que le processus va à accéder. La division en partitions s'adresse aux applications scientifiques et suppose une distribution physique adéquate pour maximiser ensuite la concurrence des accès. Une opération `open_share` permet le partage du pointeur sur la partition entre plusieurs processus de l'application.

4.2.5.2 HFS (Holographic File System) [BGF88]

HFS est un SF développé par les universités de Jérusalem et Michigan qui s'adresse aux environnements distribués qui disposent d'un grand nombre de disques. HFS évite donc tout service centralisé et est spécialement conçu pour gérer un fort dynamisme de l'architecture physique du réseau : apparition et disparition soudaines de nœuds avec disques.

HFS utilise la notion de fichiers entrelacés mais tire son nom de son approche différente dans la distribution des unités d'entrelacement d'un fichier. Plutôt que d'utiliser la politique du tourniquet, HFS distribue les enregistrements d'un fichier de façon quasi-aléatoire selon une fonction de hachage à deux critères : le nom du fichier et le nom ou l'identification de l'enregistrement. Il est important de remarquer que les unités d'entrelacement (les enregistrements) n'ont pas une taille fixe, et qu'elles peuvent donc suivre une distribution en fonction de leur utilisation.

La structure de HFS est basée sur un serveur de noms répliqué sur tous les nœuds du système et l'interface proposée est celle d'Unix.

4.2.5.3 Le SF de nCUBE [DM91, PFDJ89]

nCUBE systems offre un SF commercial développé pour son architecture nCUBE (hypercube sans mémoire commune). Ce SF est entièrement compatible avec l'interface d'Unix, même s'il permet aux utilisateurs, grâce à un utilitaire système, d'y apporter leurs propres extensions.

Les premières versions du SF de nCUBE ne comportaient pas l'entrelacement des fichiers et se limitaient à faire une distribution des fichiers dans les diverses unités de disques en fonction du préfixe de leur nom. La nouvelle version utilise une approche similaire à celle du SF précédent, avec en plus des fonctions de correspondance (*mapping functions*) qui sont en fait une composition de deux fonctions : une qui à partir d'un bloc de fichier attribue un disque où le bloc doit être stocké, et une deuxième qui à partir de l'identification d'un processus calcule le bloc du fichier qu'il doit traiter [DD93].

4.2.5.4 ELFS (ExtensibLe File System) [GL91, GP91]

ELFS est un projet de l'université de Virginie, qui s'intéresse aux limitations des E/S dans les systèmes à hautes performances, aux difficultés rencontrées par les programmeurs pour adapter leurs opérations d'E/S aux modèles de SF existants et à la prolifération de formats de données qui vont à l'encontre des caractéristiques des différentes architectures matérielles.

En suivant une approche orientée objet, ELFS propose un langage et un support système d'exécution qui permet de spécifier une hiérarchie de classes de fichiers. De nouvelles interfaces utilisateurs, associées à chaque classe de fichiers sont expérimentées. Afin de mieux s'adapter aux différents types d'applications observés, plusieurs types de fichiers (classes d'objets) sont proposés par défauts, mais l'utilisateur peut en ajouter en fonction de ses besoins. Parmi les objets définis par défaut, il se trouve un objet fichier parallèle, un objet fichier matrice, et un objet arbre binaire. Les fonctions de cache et de demande de page à l'avance peuvent être ainsi optimisées.

Le SF parallèle du projet PM [BS89] de l'université d'Erlangen en Allemagne utilise une approche similaire où les fichiers et les processus sont au même niveau dans la classification des objets.

4.2.5.5 sfs (scalable file system) [LIN93]

sfs est un SF parallèle compatible avec Unix et réalisé par le groupe "systèmes d'exploitation" de Thinking Machines. L'architecture cible est la CM-5 qui est une machine MIMD sans mémoire commune avec un sous-système d'E/S qui suit le modèle des organisations RAID de niveau 3, car on s'attend à des applications qui font des transferts de données de très grande taille (cf. section 3.4.2).

sfs utilise un système de processus disque similaire à celui de CFS, mais à une différence près : dans CFS le processus disque responsable d'un fichier ne répond qu'aux requêtes d'ouverture et

de fermeture, et une fois l'ouverture autorisée les applications envoient directement leurs requêtes de lecture/écriture aux nœuds de stockage en fonction du numéro de bloc à accéder. Dans sfs toute requête qui concerne un fichier est envoyée au processus disque responsable de la gestion de celui-ci, et c'est ce processus qui se coordonne avec les autres processus disque pour répondre à la requête. Cette solution décharge les processus de calcul de l'assemblage et du découpage des paquets d'E/S.

L'interface supportée par sfs est une extension de l'interface Unix avec de nouvelles primitives pour la manipulation directe des blocs entrelacés des fichiers. Deux caractéristiques additionnelles distinguent sfs des autres SF : le support pour des fichiers de très grande taille (pointeurs de fichiers de 64 bits) et la capacité de traiter des tailles de blocs qui ne sont pas une puissance de deux.

4.3 Conclusion

Nous avons consacré ce chapitre à l'étude de la composante logicielle principale des sous-systèmes d'E/S : le système de fichiers. Le SF est en effet essentiel ne serait-ce que parce qu'il est le seul capable de tirer le meilleur profit de l'organisation matérielle sous-jacente ; il est d'ailleurs le point crucial de notre étude.

Dès leur apparition les systèmes parallèles ont trouvé dans les systèmes répartis une source de solutions adaptables aux contraintes spécifiques du parallélisme. Rien de plus naturel alors que de se tourner encore une fois vers ce domaine pour y puiser des résultats. Cependant, bien que l'on puisse retenir quelques idées des concepts de base des SF répartis, après analyse nous observons que leur problématique est toute autre car elle reste attachée à la coexistence de plusieurs SF indépendants, alors que dans les SF parallèles le but est d'intégrer dans un seul programme parallèle toutes les entités nécessaires à la gestion des données des utilisateurs.

Divers projets de SF parallèles ont vu leur jour dans les 10 dernières années. Quelques produits commencent à être commercialisés et des projets de recherche comme Bridge et HFS ouvrent la voie qui doit nous mener à structurer correctement l'architecture logicielle des systèmes parallèles de façon à faire face au défi que représentent aujourd'hui les E/S dans les nouvelles architectures.

Deux nouveaux projets ont surgi tout dernièrement, qui s'adressent aux environnements distribués mais qui intègrent dans leurs propositions plusieurs des caractéristiques discutées au long de ce chapitre : le projet PASSION (**P**arallel **A**nd **S**calable **S**oftware for **I**nput-**O**utput) de

l’université de Syracuse [CBHK94] qui étudie les E/S d’un point de vue logiciel, et qui comprend les langages, les compilateurs, le support en exécution et VIP-FS un SF parallèle pour des environnements distribués [DBC93, DHC94] ; et le projet PIOUS (**P**arallel **I**nput/**OU**tput **S**ystem) de l’université Emory d’Atlanta [MS94] qui définie une interface au-dessus des librairies du type PVM [Su90].

Chapitre 5
Le système d'exploitation

La première partie de ce manuscrit est consacrée à l'étude des organisations matérielles des sous-systèmes d'E/S à hautes performances pour les architectures massivement parallèles ; dans la suite nous étudions une architecture logicielle capable d'exploiter au mieux les caractéristiques de ces sous-systèmes.

L'architecture logicielle des E/S concerne non seulement le système de fichiers mais aussi tout le système d'exploitation. Nous préconisons la participation du système d'exploitation, du SF et des utilisateurs, chacun à un niveau approprié, pour avoir une couche logicielle en rapport avec les diverses demandes en E/S des applications. Ainsi, ce chapitre est consacré à la composante système d'exploitation et aux fonctionnalités de ce niveau qu'il est possible d'exploiter pour améliorer les E/S.

Dans ce chapitre nous étudions le système d'exploitation et son interaction avec le SF. Nous commençons tout d'abord par une analyse de la conception actuelle des systèmes d'exploitation à base d'un micro-noyau minimal et d'un ensemble de serveurs (dont le SF). Ensuite, nous décrivons le système d'exploitation PAROS et son micro-noyau ParX développé au sein de notre équipe. Nous faisons ressortir les caractéristiques qui font de ce micro-noyau une base convenable pour la construction d'un SF universel pour les architectures massivement parallèles. En fin du chapitre nous présentons les extensions que nous apportons au modèle de ParX pour l'adapter encore mieux aux besoins des sous-systèmes d'E/S.

5.1 L'approche micro-noyau des systèmes d'exploitation

Le développement des systèmes d'exploitation modernes a suivi une approche structurée et modulaire, dite "approche micro-noyau". A l'opposé d'une conception monolithique, un micro-

noyau ne fournit qu'un ensemble de fonctionnalités et d'abstractions le plus réduit possible ; en cela il est minimal. Il constitue le coeur du système autour duquel sont construits tous les serveurs système qui offrent aux utilisateurs toutes les fonctions classiques. Une telle évolution se justifie pleinement par une plus grande portabilité, la facilité d'adjonction de nouvelles fonctions, ou encore la capacité d'adaptation à de nouvelles architectures et technologies matérielles. Les micro-noyaux sont donc une solution pour la réalisation de systèmes ouverts et distribués [Gi90].

Dans cette classe de systèmes d'exploitation le micro-noyau assure les services généraux (indépendants d'un système d'exploitation particulier), comme la gestion des entités de base telles que les processus, la mémoire, les processeurs, ou encore les objets de communication inter-processus, ceci de manière à garantir leur indépendance vis-à-vis de l'architecture matérielle sous-jacente (monoprocesseur, MIMD avec ou sans mémoire commune, etc.). Ces services de base forment un ensemble complet qui peut servir de support à toute autre fonctionnalité système spécifique.

Les services propres à chaque système peuvent être alors réalisés sur ce micro-noyau comme des serveurs système qui contrôlent d'autres ressources comme les fichiers, les dispositifs et les services de communication de haut niveau. Le SF est donc dans cette approche un serveur système qui n'est rien d'autre qu'une application utilisateur, exception faite de ses privilèges qui lui permettent d'exécuter les opérations du noyau dites sensibles.

Une architecture basée sur un micro-noyau et un ensemble de serveurs système est totalement justifiée du point de vue du génie logiciel. Bien que la taille d'un système d'exploitation structuré de cette manière soit en général augmentée, l'approche micro-noyau définie une méthodologie appropriée pour faire face à la complexité croissante dans la conception et le développement des systèmes d'exploitation modernes. Les constructeurs de systèmes ont besoin de standards sur lesquels ils puissent s'appuyer pour leurs développements ultérieurs ; et ils trouvent dans le micro-noyau les fonctionnalités de base à partir desquelles chacun peut offrir de nouveaux services tout en conservant ceux qui lui sont utiles.

Dès leur origine dans le système V [Ch88], les architectures micro-noyau ont été, et continuent à être, fortement liées aux systèmes distribués. Le développement des concepts introduits par système V s'est plutôt fait à travers des systèmes tels que Mach [Ac+86], Chorus [Ro+88], Amœba [TM82], et plus récemment Plan 9 [PPT92]. L'échange de messages qui est devenu une caractéristique courante des architectures micro-noyau, est aussi le paradigme de base des

communications dans les architectures parallèles faiblement couplées[1]. L'échange de messages offre un bon compromis entre l'autonomie des entités d'une application et une intégration globale de celles-ci : il nécessite d'une part la définition des protocoles qui précisent les règles qui doivent être respectées pour établir une communication, et d'autre part le mode d'interaction est suffisamment flexible pour ne pas empêcher le regroupement d'entités d'exécution.

La figure 5.1 schématise l'approche des architectures micro-noyau. Les serveurs interagissent avec le micro-noyau à travers une interface bien définie, et ils ont des droits spéciaux qui leur permettent d'exécuter les instructions privilégiées offertes par le micro-noyau. Hormis cela, les serveurs et les applications utilisateurs sont tous deux, vis-à-vis du micro-noyau, des processus "normaux" qui pour communiquer entre eux font appel aux primitives d'échange de messages disponibles dans le micro-noyau. Lorsqu'une application a besoin d'un service, elle envoie un message au micro-noyau (ou à un serveur prédéfini) qui, connaissant tous les serveurs disponibles dans le système, est capable d'identifier celui qui est en mesure de répondre à la requête, et donc de lui transmettre le message.

Du point de vue de l'application, le protocole d'accès aux fichiers est toujours le même : les processus utilisent l'interface offerte par le SF (`open, read, write,` etc.). La différence se trouve au niveau de l'implémentation : l'appel à une primitive est traduit par l'envoi d'un message du processus appelant au serveur de fichiers en utilisant les mécanismes de communication inter-processus du micro-noyau.

Les architectures parallèles sans mémoire commune ont été traditionnellement assimilées aux systèmes répartis, c'est pourquoi les solutions utilisées dans un domaine sont souvent adaptées pour leur réutilisation dans l'autre[2]. Il est vrai que même si les problèmes spécifiques diffèrent, les deux types de systèmes sont confrontés à l'exploitation d'une forme de parallélisme : un système réparti (ou distribué) est un réseau d'unités d'exécution qui ne partagent pas de mémoire et communiquent au moyen d'un réseau local ou plus large.

Systèmes répartis et parallèles diffèrent tout de même nettement par rapport aux aspects suivants. L'interconnexion des nœuds est une première différence : alors que les systèmes distribués sont généralement construits au-dessus de réseaux locaux à diffusion et sont en cela plus fortement couplés, les systèmes parallèles se basent sur un réseau point-à-point rapide et fiable. La notion de rapidité est ici relative et fait référence au rapport entre vitesse de traitement

---

[1] Le terme "loosely-coupled architectures" regroupe toutes les architectures multiprocesseurs sans mémoire commune. Il comprend non seulement les architectures MIMD sans mémoire commune que nous avons étudié, mais aussi les réseaux d'ordinateurs.

[2] En fait, ce transfert est beaucoup plus fréquent dans le sens systèmes répartis vers systèmes parallèles, ce qui s'explique par l'ancienneté des premiers par rapport aux deuxièmes.

des unités centrales et vitesse de communication des dispositifs physiques. L'indépendance des nœuds des sites d'un système réparti, où les sites démarrent et s'arrêtent de façon autonome, est à opposer à la coordination des nœuds dans les systèmes parallèles. De plus, s'il est fréquent qu'un système distribué soit hétérogène, ce n'est que très rarement le cas dans un système parallèle qui exploite une machine homogène. Enfin, alors que les systèmes répartis font abstraction des caractéristiques topologiques du réseau, celles-ci sont cruciales pour les machines parallèles.

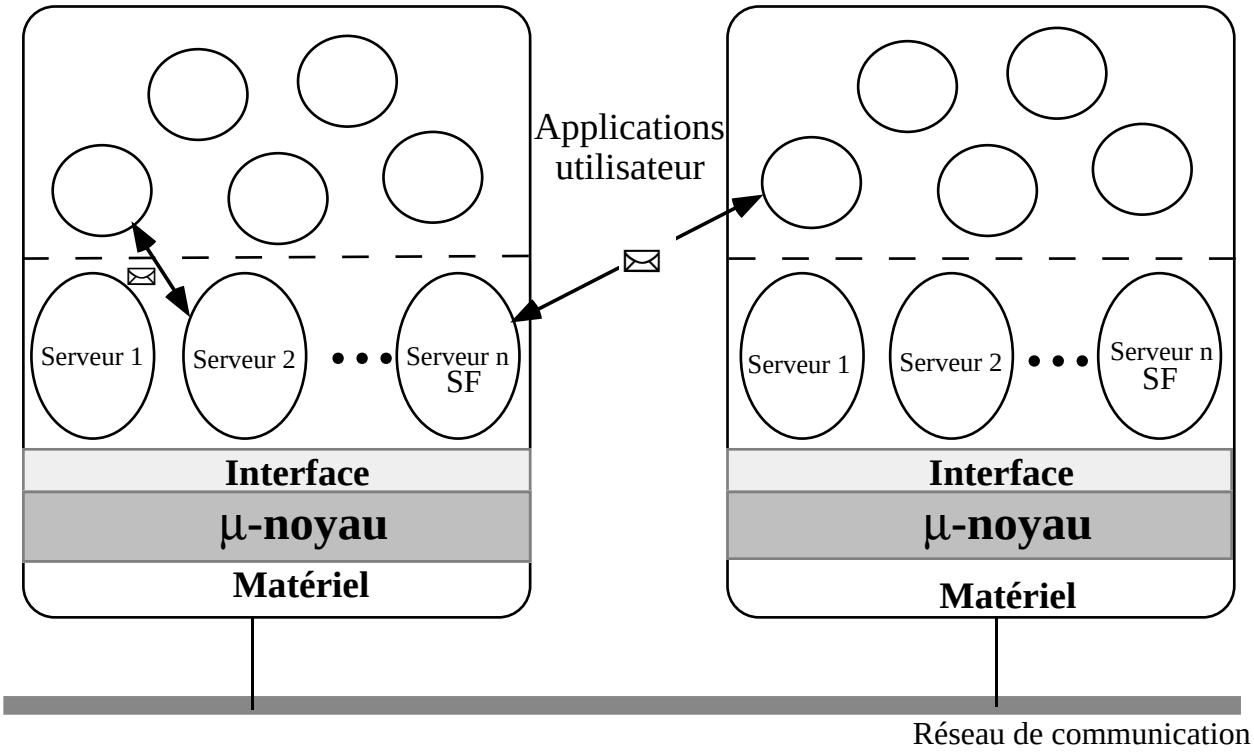


Fig. 5.1 Architecture micro-noyau.

La technologie de micro-noyau est actuellement utilisée pour les systèmes d'exploitation parallèles. Cependant pour que ce transfert des systèmes distribués vers les systèmes parallèles et massivement parallèles réussisse, des considérations spécifiques à ces systèmes doivent être prises en compte. Dans la section suivante, nous présentons PAROS, un système d'exploitation parallèle basé sur un micro-noyau spécifiquement étudié pour répondre aux besoins des architectures massivement parallèles.

5.2 Le système d'exploitation parallèle PAROS

PAROS [BCDE93, CM93, Mu94] et son noyau ParX [CEMW93, La91, Mu+89] ont été développés dans le cadre du projet ESPRIT Supernode II (P2528) [SuII91], dont l'objectif était la conception d'un système d'exploitation efficace, autonome et universel, pour machines massivement parallèles sans mémoire commune. Le but est d'atteindre de hautes performances pour l'exécution d'applications parallèles, à partir d'une construction correcte de mécanismes nouveaux pour la gestion du parallélisme, de la communication et des ressources dans des machines massivement parallèles hétérogènes.

Afin de satisfaire les applications qui nécessitent des performances élevées, et de faciliter la programmation des machines parallèles, le noyau a été conçu pour être *générique* : ParX offre en effet un mécanisme générique et correct de construction de protocoles. Implantés ainsi, les protocoles servent à la réalisation de diverses *machines virtuelles* (ou niveaux d'abstraction). Chaque utilisateur de la machine peut ainsi avoir la vision de la machine virtuelle la mieux adaptée aux besoins spécifiques de ses applications.

5.2.1 Architecture générale de PAROS

La figure 5.2 décrit l'architecture générale de PAROS. Le système a été structuré en un noyau "allégé" et un ensemble d'environnements de programmation et de sous-systèmes construits au-dessus. Le noyau ParX fournit un ensemble réduit d'abstractions de base simples et bien définies qui peuvent être utilisées par les applications et les développeurs de sous-systèmes.

ParX est découpé en différents niveaux de virtualisation grâce à des interfaces qui assurent la compatibilité des programmes avec divers modèles ou environnements de programmation, tous supportés comme des sous-systèmes (PCTE, X/OPEN, PVM, MPI, etc.). Ces sous-systèmes peuvent coexister au-dessus d'une ou plusieurs des machines virtuelles offertes par ParX, l'interface UBIK-ASI (**U**niversal **B**inary **I**nterface **K**ernel - **A**pplication **S**pecific **I**nterface) par exemple, offre un modèle de programmation mixte basé sur les variables partagées et l'échange de messages [Mu94]. Ainsi, chaque sous-système donne une vue particulière de la machine : cela pourrait être une machine Unix standard, ou bien une machine base de données ou encore une machine de calcul scientifique.

Le niveau le plus bas représente la machine virtuelle d'extension du matériel (HEM, **H**ardware **E**xtension **M**achine) qui peut être aussi bien un ordinateur parallèle à mémoire distribuée qu'un

ordinateur parallèle à mémoire commune. ParX encapsule à ce niveau toutes les caractéristiques dépendantes du matériel comme la topologie du réseau, le type de processeurs, les liens de communication, l'ordonnancement offert par le matériel, la distance entre les processeurs, etc. Grâce à des mécanismes de routage le HEM offre l'interface d'un réseau complètement connecté avec un lien virtuel entre toute paire de processeurs. Les algorithmes de routage utilisés sont corrects (sans interblocage, sans famine et équitables) et prévus pour des machines extensibles et de topologie quelconque.

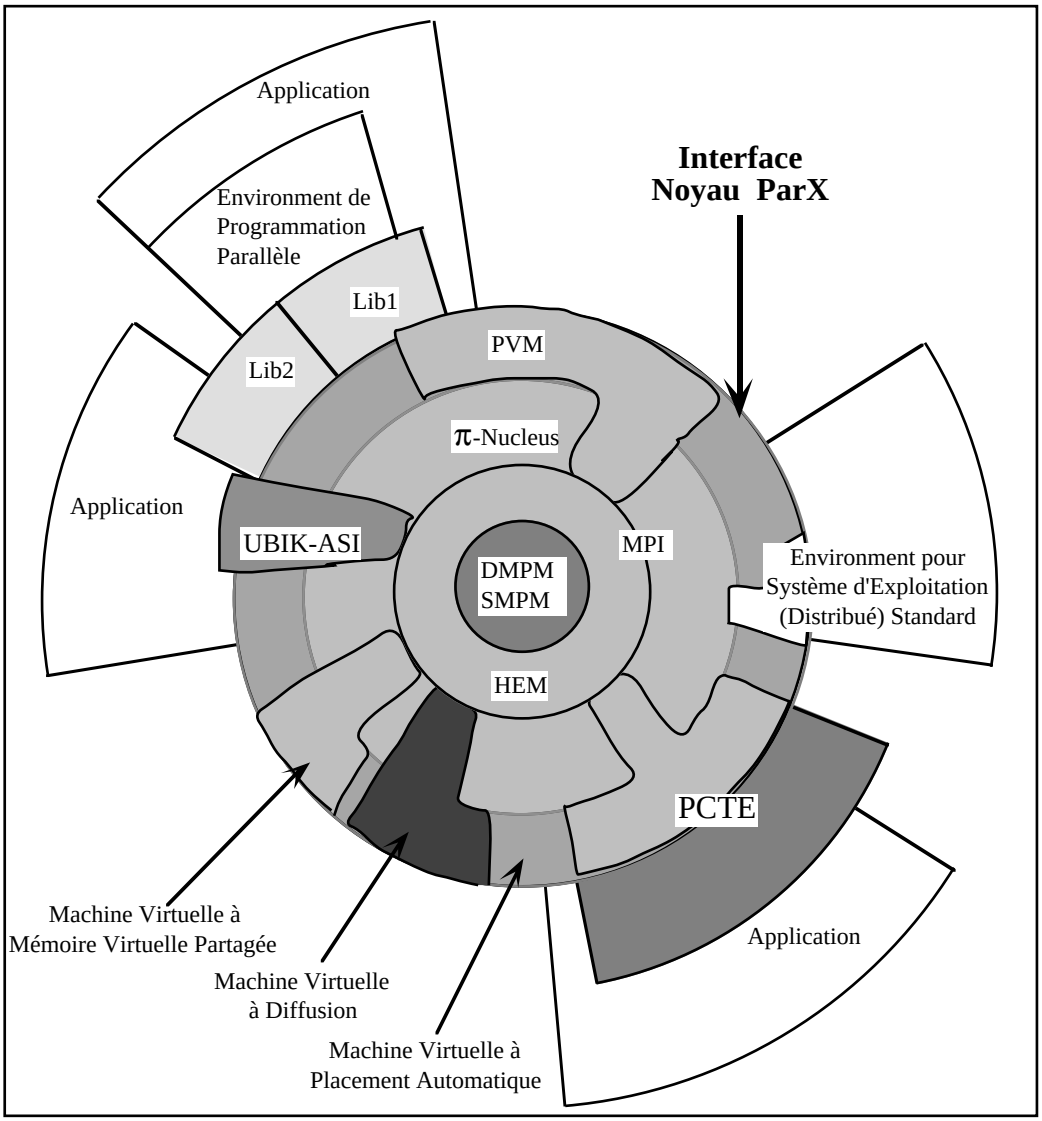


Figure 5.2 Architecture générale de PAROS.

Comme les micro-noyaux des systèmes distribués, le micro-noyau de PAROS (π-nucleus) n'offre pas les fonctionnalités traditionnelles de haut niveau des systèmes d'exploitation (gestionnaire de fichiers, accès réseau, etc.). Il fournit seulement le support essentiel à leur réalisation en tant que sous-systèmes au-dessus du micro-noyau. Le π-nucleus met en œuvre les

politiques d'allocation des ressources de base (processeurs, mémoire, etc.), et garde un contrôle sur les entités de base du système (processus et objets de communication). Des politiques d'allocation spécifiques peuvent aussi être implantées à ce niveau, pour, par exemple, assurer l'ordonnancement d'applications temps réel. Enfin, le π-nucleus offre un mécanisme générique pour la construction des protocoles utilisés par les diverses machines virtuelles (protocoles de diffusion, de rendez-vous réparti, de transfert de données distribuées, clients-serveur, etc.).

Tout autre service traditionnel, interface ou environnement de programmation peut être implanté dans les couches supérieures de PAROS. Pour cela de nouvelles machines virtuelles peuvent être ajoutées tel que nous l'avons décrit, c'est le cas par exemple des machines virtuelles à placement automatique [Ta93], à mémoire virtuelle partagée (cf. chapitre 6) et à diffusion [DM93].

5.2.2 Le noyau ParX

Dans ce paragraphe, nous donnons une brève description des concepts de base et des fonctionnalités de ParX. Trois caractéristiques fondamentales distinguent ParX des autres micro-noyaux distribués ou parallèles : le modèle de processus, le modèle de communication et la gestion de processeurs.

5.2.2.1 Modèle de processus

L'architecture générale de ParX est basée sur un modèle de processus original à trois niveaux. Il offre les concepts appropriés et le support correct pour des applications qui exploitent différents grains de parallélisme. Le modèle de processus supporte directement le concept de programme parallèle constitué de plusieurs espaces d'adressage, ainsi que le concept de flots d'instructions très légers qui s'exécutent à l'intérieur de ces espaces d'adressage. Les trois abstractions du modèle de processus sont : le *thread*, la *task* ou tâche, et la *Ptask* ou Ptâche pour tâche parallèle (cf. figure 5.3).

Un **thread** est un flot de contrôle séquentiel à l'intérieur de l'espace d'adressage d'un processeur. Les threads sont contrôlés comme des processus légers typiques : contexte d'exécution minimal, ordonnancement assuré par le matériel, etc. Ils ne permettent pas l'expression du parallélisme, mais l'utilisation de plusieurs threads à l'intérieur d'une tâche est un moyen efficace pour supporter certains constructeurs de langages parallèles (le PAR d'OCCAM par exemple), pour réaliser les programmes multi-threads (objets actifs par exemple), ainsi que pour facilement implanter des communications asynchrones au-dessus des mécanismes

de communication synchrones fournis par le noyau ParX. Le thread est une unité de pseudo-parallélisme et d'ordonnancement.

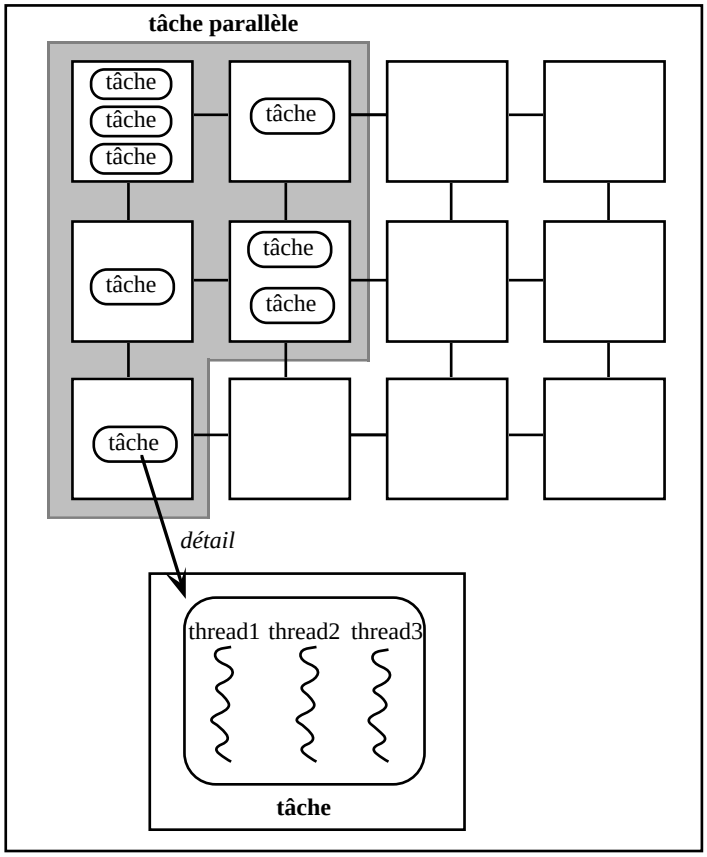


Figure 5.3 Modèle de processus de ParX.

La **tâche** est le contexte d'exécution dans lequel s'exécutent plusieurs threads. Elle est composée d'un espace d'adressage, et des structures de contrôle spécifiques à chaque application pour l'exécution et la communication de ses threads. A l'intérieur d'une tâche, les threads peuvent communiquer par mémoire partagée. Une tâche s'exécute sur un processeur logique, et elle représente une unité de parallélisme et d'ordonnancement.

La **Ptâche** représente un programme parallèle en cours d'exécution sur une machine virtuelle. Elle offre le support nécessaire à l'exécution correcte d'un programme parallèle : création et lancement de tâches, terminaison correcte selon la sémantique des différents constructeurs parallèles, contrôle de communication entre ses composantes (tâches et threads) et protection vis-à-vis des autres Ptâches dans un environnement de multiprogrammation. La Ptâche est l'entité administrative à laquelle les ressources sont allouées.

5.2.2.2 Modèle de communication

Le modèle de communication inter-processus a été soigneusement conçu pour être cohérent avec le modèle de processus. A sa base, tout protocole de communication est réalisé par des processus spécifiques qui s'échangent des messages à partir des mécanismes génériques de bas niveau fournis par le HEM [CEMW93]. Les développeurs de sous-systèmes peuvent utiliser cette interface de bas niveau pour bâtir des protocoles plus sophistiqués destinés à leur propre usage, ou encore pour construire des bibliothèques utilisateurs. A l'intérieur d'une Ptâche, les protocoles ne sont pas redondants et n'ont pas d'effets de bord sur les autres Ptâches.

Deux objets de communication de base ont été choisis pour supporter un ensemble relativement large de machines virtuelles : les **ports** et les **canaux** (cf. figure 5.4). Ils sont tous les deux *synchrones*. Le port est un objet global orienté système qui permet des communications protégées de type plusieurs vers un. Il est le mécanisme utilisé pour accéder aux sous-systèmes et aux serveurs. En revanche, le canal est un mécanisme de communication de type un vers un, entre tâches ou threads à l'intérieur d'une même Ptâche. Notons que les threads qui appartiennent à une même tâche peuvent communiquer aussi bien par mémoire partagée que sur un canal. Les canaux sont locaux à l'intérieur d'une Ptâche et ne peuvent être utilisés comme moyen de communication entre Ptâches ; le seul moyen de communication entre Ptâches est le port .

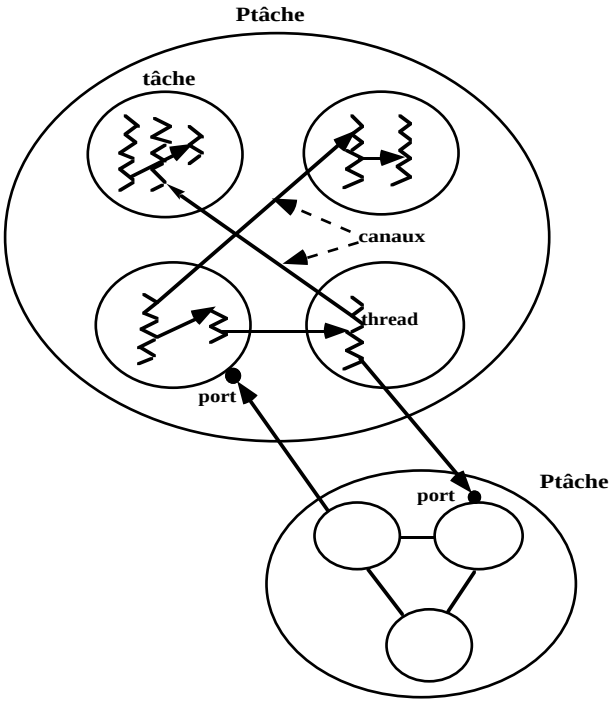


Figure 5.4 Communications entre Ptâches, tâches et threads.

5.2.2.3 Gestion des processeurs

Comme le montre la figure 5.5, deux types de regroupement de processeurs sont à distinguer : un *bouquet* de processeurs physiques nus (**bunch**) ou une *grappe* d'espaces d'adressage disjoints (**cluster**) localisés sur un ensemble connexe de processeurs du réseau (statiquement ou dynamiquement configurable).

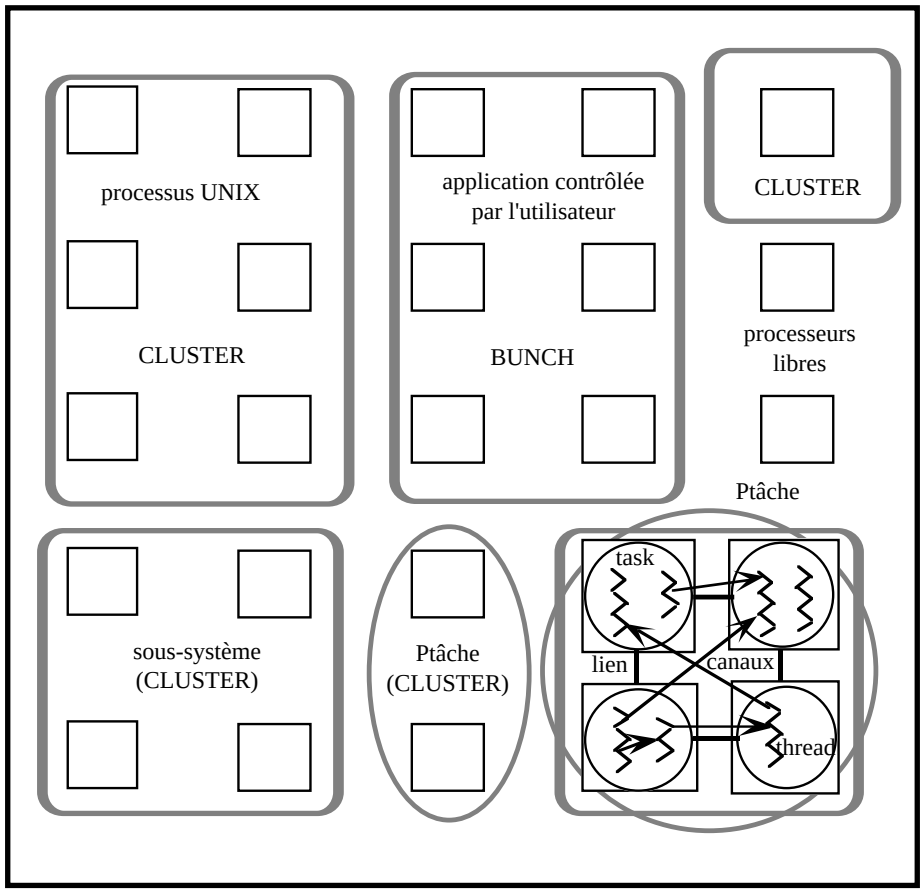


Figure 5.5 Bunchs et clusters dans ParX.

Bunch

Un bunch est un ensemble de processeurs et de connexions entre ces processeurs qui permet à l'utilisateur l'exploitation directe d'une "machine nue". De cette façon PAROS peut être rendu compatible avec les actuels environnements parallèles de bas niveau (C 3L, C INMOS dans le cas du transputer).

Une application parallèle existante qui s'exécute sur une machine nue sera directement portable sur un bunch sans avoir à la réécrire. De plus, certains utilisateurs préféreront le bunch afin

d'exécuter des programmes parallèles avec un maximum d'efficacité sans payer le surcoût nécessairement introduit par les fonctionnalités offertes par le noyau d'un système.

Cluster

C'est la machine virtuelle offerte pour l'exécution des programmes parallèles, et qui est constituée d'un ensemble de nœuds avec une configuration donnée et d'un support noyau de système complet (gestion des ressources, protocoles spécifiques de communication et de synchronisation, domaine de protection, etc.). Chaque nœud est constitué d'un ou plusieurs processeurs et d'une mémoire locale.

La fonction principale d'un cluster est d'offrir un support physique de telle sorte que seule la structure logique d'une tâche parallèle ait besoin d'être spécifiée. Le programme correspondant sera placé par le système ou par l'utilisateur sur les ressources physiques disponibles au moment du chargement.

Le cluster est une entité dynamique dans laquelle les nœuds physiques peuvent être alloués et libérés durant l'exécution de la Ptâche. Cette propriété est utile pour améliorer l'utilisation des ressources inoccupées, pour supporter l'implantation de mécanismes de tolérances aux pannes, etc.

5.2.3 Les clusters de gestion

Nous avons vu, dans la section précédente que sur chaque nœud de la machine parallèle s'exécute le micro-noyau (exception faite des bunchs) qui offre différents niveaux de machines virtuelles. Au-dessus, des sous-systèmes fournissent un environnement de développement et d'exécution des applications, chaque système mettant en œuvre un ensemble de politiques qui lui est propre, à partir des mécanismes de base fournis par le micro-noyau.

L'existence simultanée de sous-systèmes très divers nécessite un certain nombre de clusters sur lesquels s'exécutent des Ptâches dont le rôle est la gestion et l'administration des Ptâches utilisateurs [Me93]. La figure 5.6 donne un aperçu général de la gestion d'une machine parallèle dans PAROS. Nous expliquons plus en détail, dans les sections suivantes, le rôle des différents clusters et des différentes Ptâches du système.

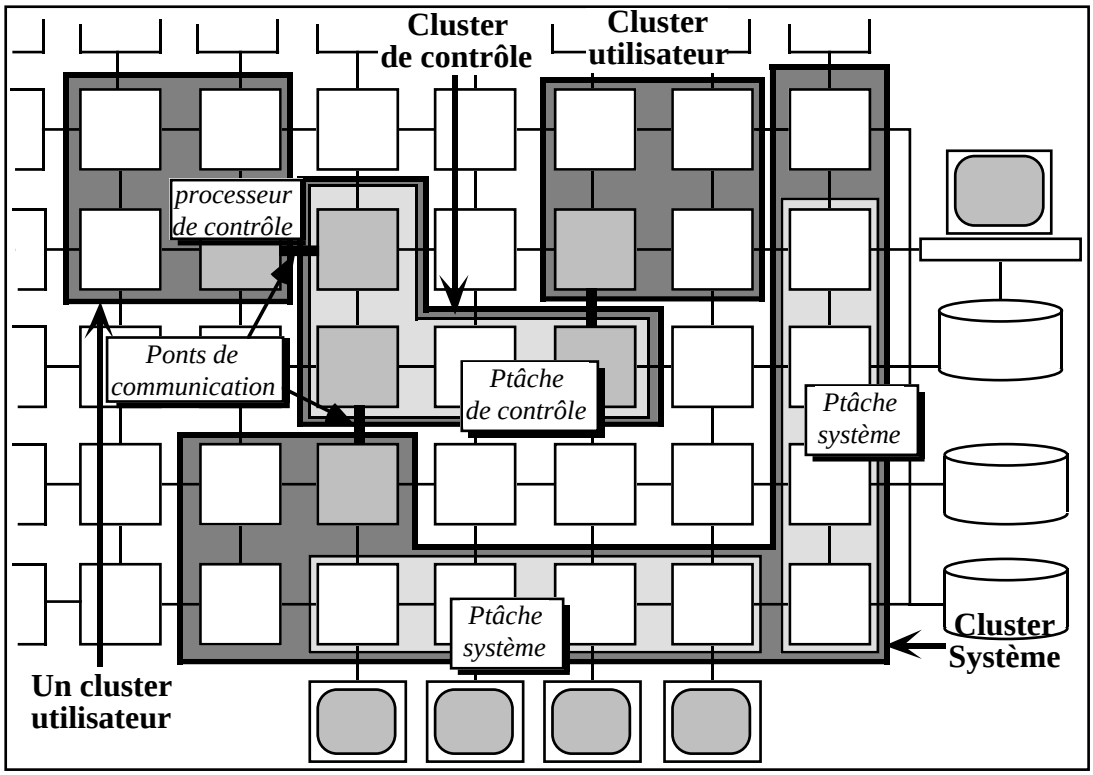


Figure 5.6 Gestion d'une machine parallèle dans PAROS.

5.2.3.1 Le cluster de contrôle

Le cluster de contôle a pour rôle la gestion des Ptâches utilisateurs, l'interface avec le système et la communication entre Ptâches. Comme ce cluster n'est constitué que d'une seule Ptâche, nous la désignerons sous le nom de *Ptâche de contrôle.* Chaque fois qu'une Ptâche est lancée, le *chargeur* (qui est un service assuré par le système) choisi un sous-ensemble de nœuds de ce cluster pour assurer la gestion de la nouvelle Ptâche. Les nœuds du cluster utilisateur qui sont directement reliés aux processeurs du cluster de contrôle deviennent alors les *nœuds de contrôle* de gestion de la Ptâche. Ces liaisons directes entre le cluster utilisateur et le cluster de contrôle sont nommées *ponts de communication*.

Dans ParX, un cluster fournit un domaine de communication dans lequel chaque nœud est capable d'envoyer un message vers n'importe quel autre nœud du cluster. Pour toute communication à l'extérieur du cluster, les messages sont envoyés vers l'un des nœuds de contrôle de la Ptâche pour que celui-ci les transmette, à travers les ponts de communication, à la Ptâche de contrôle. Comme cette dernière dispose d'un pont de communication avec toutes les Ptâches en exécution, elle peut, en particulier, transmettre les messages aux nœuds de contrôle de la Ptâche destination.

5.2.3.2 Le cluster système

Le cluster système est le nerf du système ; il est composé de plusieurs Ptâches que nous appelons *Ptâches systèmes* par opposition aux Ptâches utilisateurs. La figure 5.7 montre la composition du cluster système dans PAROS :

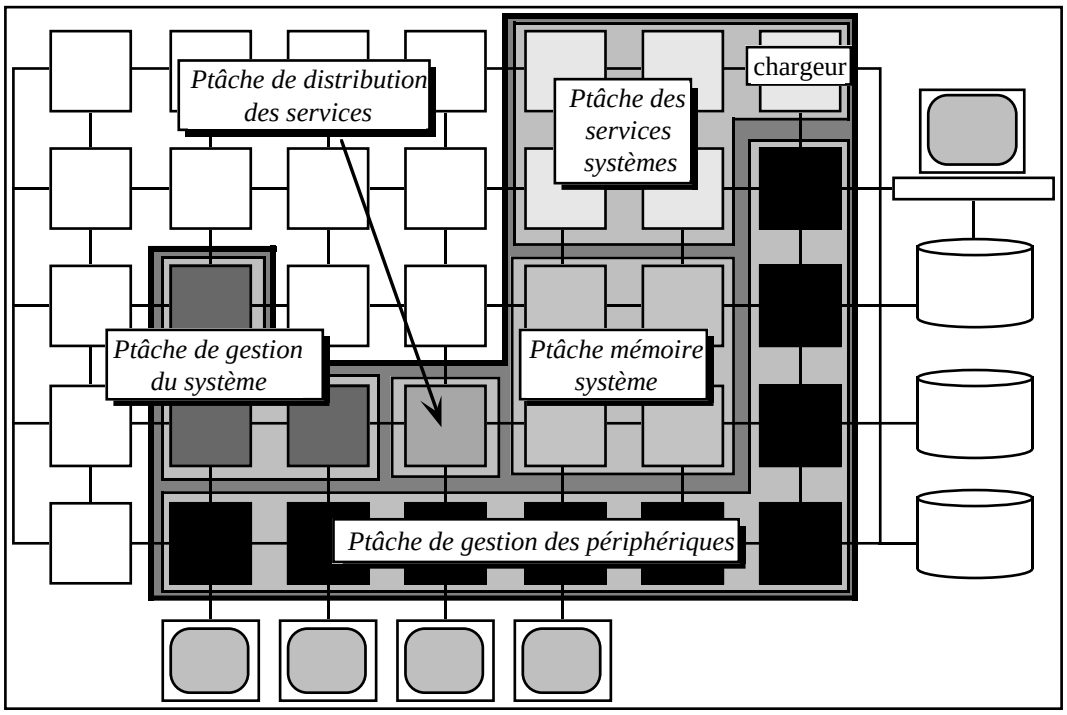


Figure 5.7 Le cluster système.

- la *Ptâche de gestion du système* enregistre la localisation et l'état (suspendue, en cours d'exécution, etc.) de chacune des Ptâches de la machine, garde la trace des différents environnements et des différents périphériques disponibles (disques, écrans, imprimantes, etc.), et tient à jour l'état des ressources disponibles (processeurs, mémoire, ports, etc.) ;

- la *Ptâche des services système* contient des programmes systèmes tels que le *chargeur d'applications*, l'*allocateur d'entités aux processeurs*, l'*outil d'équilibrage de charge*, *l'outil de reconfiguration de la machine* lorsque cette fonctionnalité est disponible, etc. ;

- la *Ptâche de pilotage des périphériques* comme son nom l'indique s'occupe de la gestion des périphériques. Pour qu'un périphérique puisse être accédé par toutes les Ptâches utilisateurs, il faut qu'il soit attaché à un processeur de cette Ptâche. Ainsi, tous les processeurs d'E/S sont alloués à cette Ptâche pour faire des unités de stockage une ressource partagée par toutes les Ptâches du système ;

- la *Ptâche mémoire système* sert de zone mémoire tampon pour les différentes Ptâches du cluster système (par exemple pour stocker temporairement une Ptâche ou un fichier en cours de chargement) ;

- la *Ptâche de distribution de services* sert uniquement à recevoir les requêtes qui viennent des Ptâches du "monde extérieur" (i.e. qui ne sont pas dans le cluster système) et à les rediriger vers les services correspondants (Ptâche de pilotage de périphériques, gestionnaire des Ptâches, etc.). Cette Ptâche est très utile lorsqu'il n'est plus possible

d'atteindre directement un service système donné, par exemple parce que tous ses accès directs sont utilisés.

5.3 Extensions apportées au modèle ParX

ParX est un noyau de système d'exploitation où les concepts du parallélisme sont entièrement intégrés, ce qui fait de lui un micro-noyau adapté aux systèmes massivement parallèles. La gestion des E/S pose néanmoins certains problèmes qui n'ont pas été abordés lors de la présentation précédente. Dans cette section nous identifions ces problèmes et proposons trois types d'extensions au micro-noyau ParX afin de mieux l'adapter aux besoins spécifiques des sous-systèmes d'E/S. Nous développons ces extensions au niveau du modèle de processus, de la gestion de processeurs et de l'implémentation des protocoles de communication.

5.3.1 Modèle de processus

La résolution d'un problème peut nécessiter la coopération de deux ou de plusieurs applications parallèles indépendantes, ou la création dynamique de Ptâches au milieu de l'exécution d'une autre pour résoudre un sous-problème spécifique du problème original (cas typique de l'appel à une bibliothèque parallèle). L'exécution de ces Ptâches dans le même cluster peut faciliter la coopération et améliorer ainsi les performances globales des applications. Il faut noter que le partage d'un cluster par deux ou plusieurs applications n'implique pas le partage des nœuds à l'intérieur du cluster. Chaque Ptâche peut avoir son sous-ensemble propre de nœuds du cluster.

Le partage de fichiers est une forme répandue de coopération entre applications et c'est au sous-système d'E/S qui revient la tâche d'assurer la cohérence de l'information des fichiers accédés de façon concurrente. Pour la gestion de cette cohérence nous devons tout d'abord définir l'entité système à l'intérieur de laquelle ce partage de fichiers a lieu, c'est là l'objectif de notre première extension du modèle ParX.

Nous considérons que des Ptâches qui partagent des fichiers sont des Ptâches qui coopèrent et de ce fait, le système devrait leur allouer un seul cluster pour leur exécution. Ceci n'a pas de conséquence sur le nombre de processeurs alloués à l'ensemble des Ptâches car le nombre de processeurs alloués à un cluster partagé par plusieurs Ptâches peut être le même que si l'on avait alloué un cluster par Ptâche.

La possibilité d'exécuter plusieurs Ptâches dans un même cluster avait déjà été introduite dans la définition du cluster système de PAROS de la section 5.2.3.2. Cependant, comme nous allons le voir, pour que l'exécution de plusieurs Ptâches dans un seul cluster soit efficace, la définition de la Ptâche doit être modifiée, et le modèle de processus de ParX doit être revu. En effet, la définition de la Ptâche lui attribue le rôle de domaine de communication et de protection ce qui garantit que les entités à l'intérieur d'une Ptâche puissent communiquer entre elles sans être perturbées par des communications externes. Pour la communication entre différentes Ptâches, le système prévoit l'existence de *ponts de communication* pour acheminer les messages d'une Ptâche à une autre (cf. section 5.2.3.1). Or, ces ponts de communications sont prévus en réalité pour la communication inter cluster, c'est pourquoi ils sont reliés à la Ptâche de contrôle. La communication directe entre Ptâches à l'intérieur d'un cluster n'est alors pas possible !

La réalisation de ParX nous permet de mieux comprendre le problème. L'acheminement de messages entre une paire de processeurs à l'intérieur d'un cluster est assuré par une table de routage dont chaque routeur dispose de la partie qui lui correspond [Go91]. Cette table est associée à tout le cluster car elle est calculée lors de la création du cluster. C'est alors le cluster, et non pas la Ptâche, qui fournit le domaine de communication naturel au système.

Il est néanmoins nécessaire d'avoir une entité logique, supportée directement par le micro-noyau, à laquelle associer ce rôle de domaine de communication et de protection, et à l'intérieur de laquelle le système garantit la cohérence des informations manipulées. Nous proposons d'étendre le modèle de processus avec un niveau supplémentaire, *la ligue*[1], et de relever la Ptâche de ces attributions.

Une **ligue** est composée de l'ensemble des Ptâches qui s'exécutent sur un même cluster. Elle définit un domaine de communication et de protection vis-à-vis des autres ligues. Les communications à l'intérieur d'une ligue se font en respectant les définitions du modèle de communication (p. ex. un canal ne peut pas être utilisé pour communiquer entre Ptâches différentes) (voir figure 5.8). De plus, toute communication entre ligues devra passer par l'intermédiaire du système (Ptâche de contrôle). L'association entre ligue et cluster est bidirectionnelle, c'est-à-dire, une ligue ne s'exécute jamais sur plus d'un cluster et un cluster n'exécute jamais plus d'une ligue[2].

Le chargement d'une Ptâche dans ParX supposait l'existence d'un cluster déjà alloué à l'utilisateur. Avec la ligue, l'existence d'un cluster est lié à l'existence d'une ligue et de ce fait les deux entités sont créées en même temps. Dans le fichier de description d'une ligue utilisé par

[1] Le terme ligue a été choisi parce qu'il avait déjà été introduit par le système PEACE [Sc90] pour faire référence au domaine de protection de l'ensemble d'applications d'un utilisateur.

[2] D'ailleurs, le deux termes sont sous-entendus équivalents dans la suite.

le chargeur PAROS, l'utilisateur doit alors spécifier non seulement les Ptâches qui composent sa ligue, mais aussi les bornes inférieures et supérieurs du nombre de processeurs qui doivent être alloués au cluster (cf. thèse d'Yves Langué [La91], section 6.1.1, page 74 sur la création des clusters). Les Ptâches sont chargées et lancées dans l'ordre déterminé par l'utilisateur, et chaque Ptâche s'exécute indépendamment des autres. La ligue maintient l'état de toutes ses Ptâches ; les communications entre les Ptâches d'une même ligue n'ont pas besoin d'utiliser les ponts de communication. La terminaison de la dernière Ptâche d'une ligue entraîne la destruction de la ligue et la libération des processeurs du cluster.

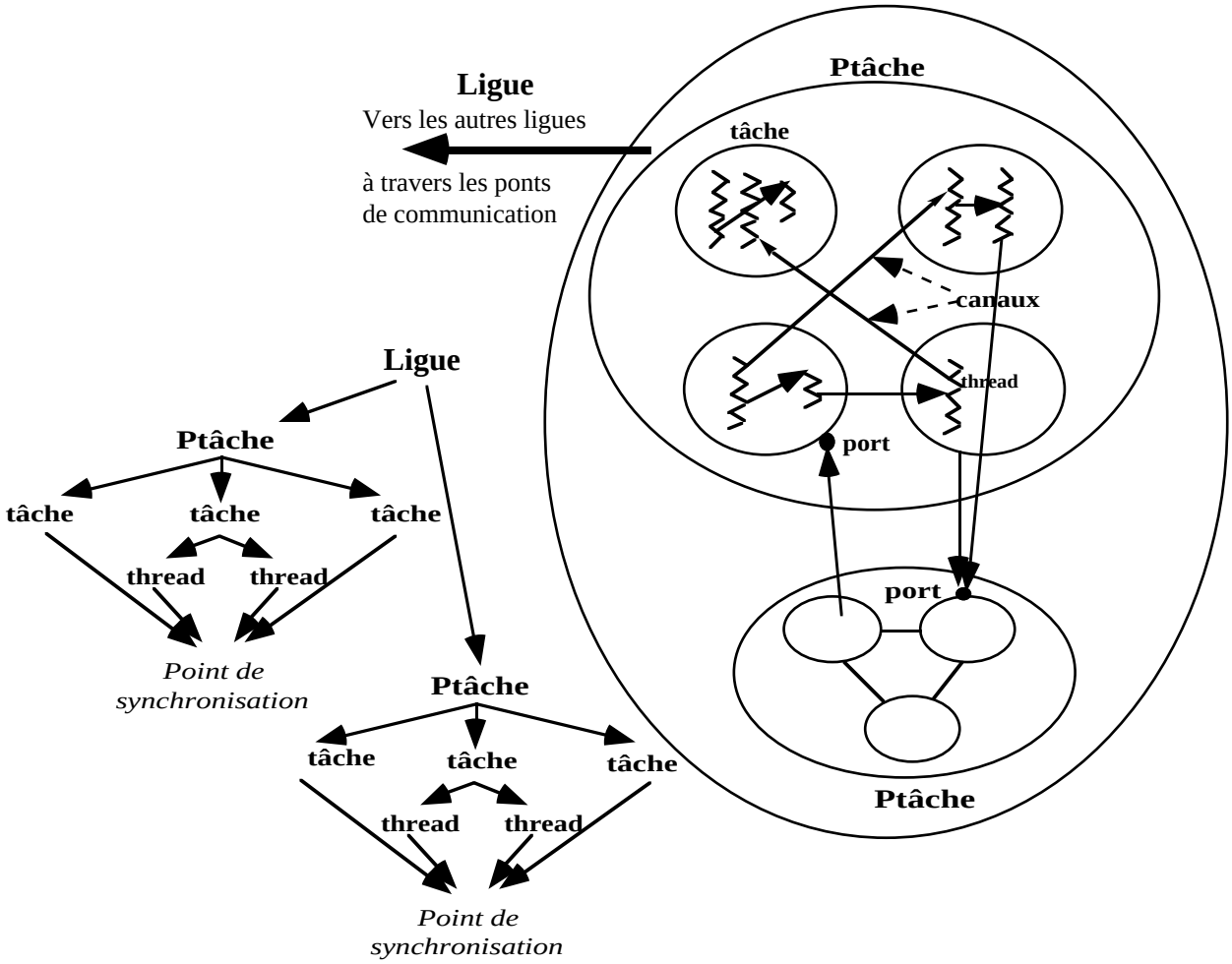


Figure 5.8 Modèle de processus étendu.

Chaque Ptâche reçoit un ensemble privé de processeurs du cluster, d'où la protection de l'exécution d'une Ptâche par rapport aux autres Ptâches de la ligue. L'allocation des ressources est alors partagée entre la ligue et ses Ptâches. Ainsi, un cluster est alloué à une ligue, mais les processeurs à l'intérieur d'un cluster sont alloués aux Ptâches de la dite ligue. Des appels de création de Ptâches permettent à une ligue de charger dynamiquement de nouvelles Ptâches sur le cluster. En outre, lorsqu'une seule Ptâche s'exécute sur un cluster, elle peut être assimilée à une ligue. Ainsi pour faire référence à la ligue qui s'exécute sur le cluster de contrôle nous

continuons à utiliser le terme Ptâche de contrôle. Nous définissons de plus la ligue système comme l'ensemble des Ptâches qui occupent le cluster système.

Puisque la ligue représente le domaine de protection et de communication, le partage de fichiers, et par conséquent la gestion de la cohérence, sera fait au niveau de la ligue. Pour que deux Ptâches puissent accéder simultanément à un fichier, et bénéficier de la gestion de cohérence offerte par le système, il faut qu'elles appartiennent à la même ligue. Deux Ptâches qui appartiennent à des ligues différentes ne disposent alors d'aucun contrôle système pour leurs accès aux fichiers.

Limiter la gestion de la cohérence à l'intérieur d'une ligue n'est pas une contrainte pénalisante car il est naturel que deux Ptâches, si elles coopèrent, appartiennent à la même ligue. Ainsi, si l'on considère le partage de fichiers comme une forme de coopération, la gestion de la cohérence peut être placée à l'intérieur de ce domaine de coopération. Par ailleurs, si deux Ptâches ne nécessitent pas le partage d'information, elles peuvent s'exécuter sur des clusters différents.

5.3.2 Le cluster mémoire

Les E/S posent deux problèmes fondamentaux au gestionnaire de mémoire d'une machine parallèle à mémoire distribuée. D'une part, afin de limiter le nombre d'accès aux unités de stockage, le sous-système d'E/S a besoin d'un certain nombre de tampons système pour maintenir les données qui risquent d'être consultées de nouveau (*caching*) et pour en précharger d'autres afin d'anticiper leur demande (*prefetching*). D'autre part, comment effectuer, d'une manière efficace à travers l'ensemble de mémoire privées, le vidage sur disque de certaines données en mémoire afin de gagner de la place pour en installer de nouvelles ? (problème de "swap"). Or, si dans les systèmes traditionnels à architecture monoprocesseur, ces techniques ne sont pas trop pénalisantes, elles peuvent devenir extrêmement problématiques sur des machines parallèles. En effet, dans une machine massivement parallèle la mémoire à purger peut éventuellement traverser un certain nombre de nœuds avant de se retrouver sur le disque, et les tampons système, du fait de leur taille, peuvent être impossibles à implémenter dans la mémoire locale des processeurs d'E/S.

Un moyen d'y remédier consiste à créer un ou plusieurs serveurs mémoire qui fassent office de cache pour le disque. Il est alors possible d'établir une sorte de hiérarchie mémoire qui peut être exploitée, par exemple, par un système de pagination de mémoire virtuelle. C'est le rôle du cluster mémoire (ou ligue mémoire). Le cluster mémoire offre donc une zone de mémoire auxiliaire aux applications.

Puisque les clusters sont alloués exclusivement aux ligues, un cluster mémoire est associé à une seule ligue. Inversement, une ligue ne peut demander que la création d'un cluster mémoire. Pour optimiser son efficacité, en augmentant le débit de transfert des données entre une ligue et son cluster mémoire, le cluster mémoire dispose d'un pont de communication direct avec sa ligue. Il est relié au cluster système par la Ptâche ou ligue de contrôle (figure 5.9).

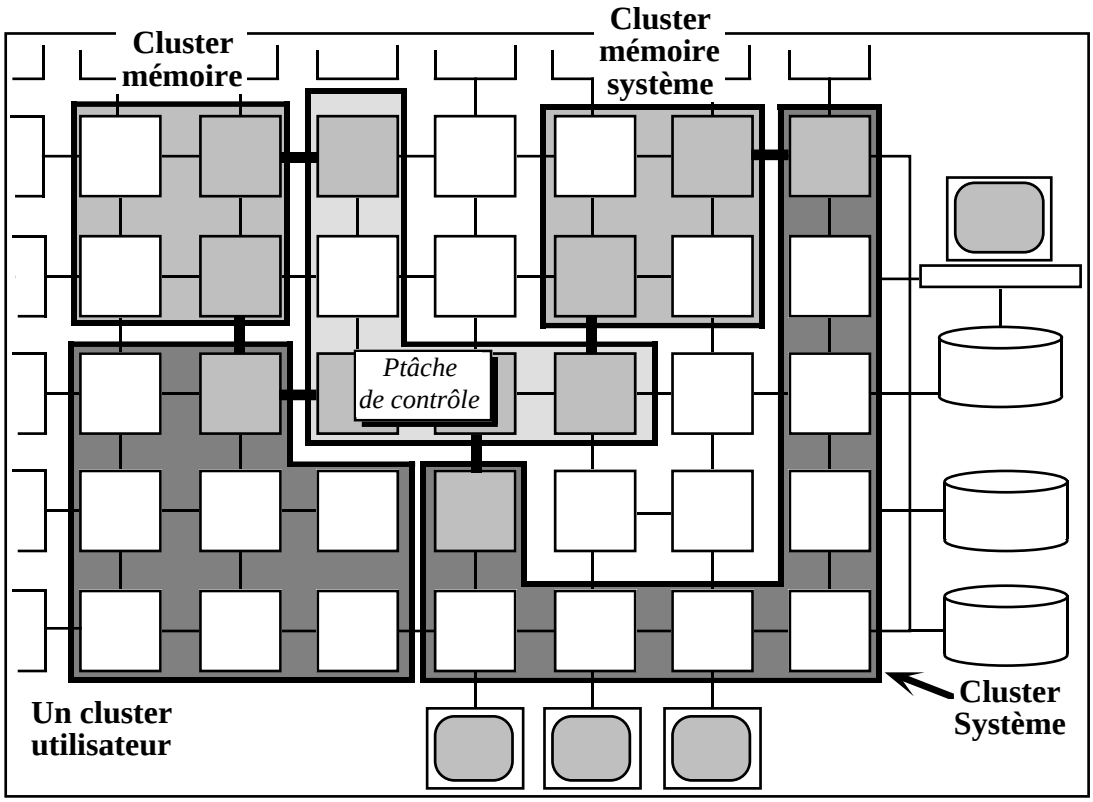


Figure 5.9 Le cluster mémoire

Un cluster mémoire peut être créé par la primitive :

```
t_error cluster_mémoire_create (    t_cluster *id_cluster,
                                    int       taille_demandée
                                    int       *taille_obtenue
                                )
```

Le paramètre `id_cluster` sert à retourner une capacité désignant le cluster créé. `taille_demandée` est la taille souhaitée de la zone de mémoire auxiliaire associée à la ligue. Le système calcule ensuite le nombre de processeurs nécessaires pour réserver cette taille mémoire et crée le cluster correspondant. `taille_obtenue` sert à retourner la taille mémoire disponible dans le nouveau cluster mémoire. Il se peut que celle-ci soit inférieure à la taille demandée si le système ne dispose pas d'assez de ressources libres.

La taille de la mémoire d'un cluster mémoire peut être modifiée au cours de son existence.

```
t_error cluster_mémoire_resize (    t_cluster id_cluster,
                                    int       taille_demandée
                                    int       *taille_obtenue
                                )
```

permet de modifier la taille de la mémoire allouée par le cluster mémoire `id_cluster`. `taille_demandée` et `taille_obtenue` ont la même sémantique que dans la primitive de création.

Enfin l'opération qui permet de détruire un cluster mémoire est la même que celle d'un cluster utilisateur :

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

Dès sa création un cluster mémoire est associé à une ligue utilisateur ; ce n'est donc pas une ressource "partageable" avec les autres ligues, et son identification n'est connue que par la ligue utilisatrice qui demande sa création (ainsi que par la ligue système, bien entendu). La création d'un cluster mémoire est une possibilité offerte à toutes les ligues. Parmi les diverses ligues, la ligue système peut aussi créer un cluster mémoire ; c'est le seul cas où les données stockées peuvent appartenir à différentes ligues. La Ptâche mémoire système s'exécutera alors sur ce cluster.

Le cluster mémoire est géré par la Ptâche mémoire (seule Ptâche qui s'exécute dans le cluster mémoire). Cette Ptâche offre deux services à l'extérieur : d'une part elle stocke des données dans son cluster à la demande de la ligue système et de la ligue utilisatrice associée au cluster mémoire ; et d'autre part, elle renvoie ces données à sa ligue utilisatrice lorsque celle-ci le lui demande. La gestion du cluster mémoire est présentée plus en détail au chapitre suivant lors de la discussion sur la gestion mémoire.

5.3.3 Transfert de gros messages

Le trafic de messages généré par les opérations d'E/S est suffisamment important pour que, dès la conception des machines, il faille penser à leur dédier un réseau spécifique afin de ne pas trop saturer le réseau de communication inter- processeurs (cf. section 2.3.1). Puisque le partage de ce réseau de communication est inévitable dans la plupart des architectures, il faut étudier la façon de diminuer l'impact des E/S sur les performances du système de communication.

L'encombrement du réseau de communication vient du fait que les messages transférés par les opérations d'E/S sont de très grande taille. Les échanges de messages entre les différents processus d'une application parallèle sont en général de petite taille, alors qu'une requête d'E/S peut nécessiter le transfert de tout un fichier. Ce phénomène n'est pas spécifique aux E/S, on l'observe aussi dans le cas transfert de l'image d'un processus lors de sa migration [El94].

Il fallait alors doter ParX d'un protocole spécifique pour le transfert de gros messages. Nous avons donc participé à cet effort dont le résultat est la réalisation d'un protocole basé sur l'utilisation de chemins alternatifs pour l'acheminement de gros messages et une étude de la viabilité d'utiliser la compression de données comme mécanisme pour accélérer leur transfert. Pour l'implémentation de ce protocole et des algorithmes de compression nous nous sommes basés sur la version actuelle de ParX, qui s'exécute sur une machine Supernode à base de transputers du type T800. C'est pourquoi plusieurs aspects de la discussion qui suit sont spécifiques à cette implémentation, cependant l'utilisation des chemins alternatifs et/ou de la compression de données est envisageable quel que soit le support système disponible.

5.3.3.1 Les chemins alternatifs

Dans le cas des architectures parallèles à réseaux d'interconnexion maillés, il peut exister plus d'un chemin entre deux processeurs ; il serait alors utile de découper un message suivant le nombre de chemins alternatifs et d'envoyer les différents morceaux en parallèle à travers ces chemins. En effet, ce mécanisme permet d'augmenter la bande passante pour l'envoi de messages entre deux processeurs et de mieux répartir la charge de communication sur le réseau.

ParX a été conçu de façon à ce qu'il puisse s'exécuter sur des machines parallèles où les processeurs sont interconnectés par un réseau de topologie quelconque. Pour cela, deux algorithmes de génération des chemins de routage, à partir de la description du réseau d'interconnexion, ont été développés dans notre équipe [Go91, Mu93].

L'implémentation actuelle de ParX utilise un algorithme de calcul des chemins de routage basé sur la génération d'un arbre recouvrant qui englobe tous les processeurs d'un cluster et un sous-ensemble des liens de communication. Les messages sont alors acheminés entre les processeurs le long des branches de l'arbre et des passerelles définies entre ces branches. Cet algorithme garantit l'absence d'interblocage [Go91].

Dans sa version déterministe l'algorithme de calcul des chemins de routage consiste à :

Etape 1. Calculer l'arbre recouvrant.

Etape 2. Déterminer le plus court chemin entre chaque paire de processeurs en tenant compte des restrictions imposées par les règles de routage pour éviter les interblocages.

Afin de pouvoir exploiter des chemins alternatifs entre chaque paire de processus, l'étape 2 a été modifiée de façon à ce que l'algorithme puisse générer le maximum de chemins alternatifs qui respectent les règles de routage pour éviter les interblocages. De plus, il est nécessaire que les chemins soient disjoints, c'est-à-dire, qu'ils n'aient pas d'arête en commun. Notons que le nombre de chemins alternatifs est forcément inférieur au degré d'interconnexion des processeurs émetteur et récepteur.

Le débit de communication entre une paire de processeurs varie selon le nombre de chemins alternatifs générés par l'algorithme. Ce nombre dépend de la topologie du réseau et de l'arbre de recouvrement retenu à l'étape 1. L'implémentation de ce protocole par A. Elleuch [El94] au-dessus de la couche de routage de ParX, a montré que lorsque la taille du message à transférer est importante on obtient un gain considérable, proportionnel au nombre de chemins alternatifs[1] :

$$V_a = V_1 \times a$$

où V_a est la bande passante maximale lorsqu'il existe *a* chemins alternatifs.

5.3.3.2 La compression de données

Depuis longtemps la compression a été utilisée comme un moyen pour accélérer le transfert de messages entre deux ordinateurs sur un réseau local ou de grande distance [He84]. Dans cette section nous étudions l'utilité et la faisabilité d'un tel mécanisme pour le transfert de messages inter-processeurs à l'intérieur d'une machine parallèle.

L'intérêt de la compression comme mécanisme pour réduire l'encombrement du réseau d'interconnexion est évident : diminuer la taille des messages qui circulent sur le réseau, sans pour autant perdre de l'information, diminue le trafic global et donc le risque de contention entre les messages. Or, pour que l'utilisation de la compression soit possible comme complément d'un mécanisme d'acheminement de messages, il faut que celle-ci ne retarde pas la délivrance des messages, c'est-à-dire, qu'elle ne diminue pas le débit de transfert, ni augmente la latence des communications. Elle doit au contraire, diminuer le temps de transfert des messages.

La caractéristique principale du mécanisme sera alors sa rapidité, aussi bien pour la compression que pour la décompression. Nous avons donc choisi la famille d'algorithmes LZRW* de R.N. Williams qui sont les algorithmes les plus rapides pour ces opérations [Wi91]. Par ailleurs, la

[1] Pour les messages de grande taille, le temps de transfert domine largement le temps du découpage et de reassemblage des messages.

mémoire nécessaire pour leur exécution est constante (16K pour chaque opération), ce qui permet d'envisager leur implémentation au niveau du micro-noyau. Leur taux de compression varie selon les données d'entrée entre 30% et 80%, avec une moyenne de 50%.

Nous avons implémenté ces algorithmes sur le transputer pour comparer leurs performances avec celles de la couche d'acheminement de messages de ParX. Nous avons testé la compression et la décompression sur plusieurs types de données et nous avons pu observer que même dans les meilleurs cas, les débits de compression et décompression sont inférieurs aux débits d'acheminement de messages de ParX. L'utilisation de chemins alternatifs ne fait qu'augmenter cette différence. Le coût d'exécution des opérations de compression et de décompression ne permet donc pas d'envisager leur utilisation pour améliorer le temps de transfert d'un message car le débit de transfert serait limité par les débits de la compression et la décompression. Le tableau 5.1 montre, pour un cas favorable de compression, ces débits (en Kilo-octets/s) et le pourcentage de compression obtenu (nouvelle taille / ancienne taille du message).

% compression des données	Débit de compression	Débit de décompression	Débit de transfert sans chemins alternatifs	Débit de transfert avec 4 chemins alternatifs
13.8 %	417 K-o/s	317 K-o/s	680 K-o/s	2720 K-o/s

Tableau 5.1. Débits de transfert, de compression et de décompression.

Ces résultats ne sont pas étonnants surtout que dès le départ, le fait d'utiliser le processeur pour faire un calcul nécessaire au routage (la compression et la décompression des messages), va à l'encontre de la conception des architectures parallèles modernes. En effet, les nouvelles machines sont dotées des unités matériel de routage qui, combinées à des mécanismes de DMA, permettent que les communications inter-processeurs d'une machine parallèle s'exécutent de façon indépendante au calcul.

L'idée de compression reste néanmoins valable, si l'on s'appuie sur le matériel pour sa réalisation. Ils existent aujourd'hui des circuits avec des débits de compression et de décompression qui assurent jusqu'à 40 Méga-octets/s (le circuit ALDC-MACRO d'IBM par exemple). Ces circuits sont bien adaptés pour les machines parallèles parce qu'ils utilisent des mécanismes de DMA, ne nécessitent pas de mémoire externe, et disposent des interfaces nécessaires pour leur intégration aux processeurs de calcul. La figure 5.10 montre l'organisation fonctionnelle d'un tel circuit

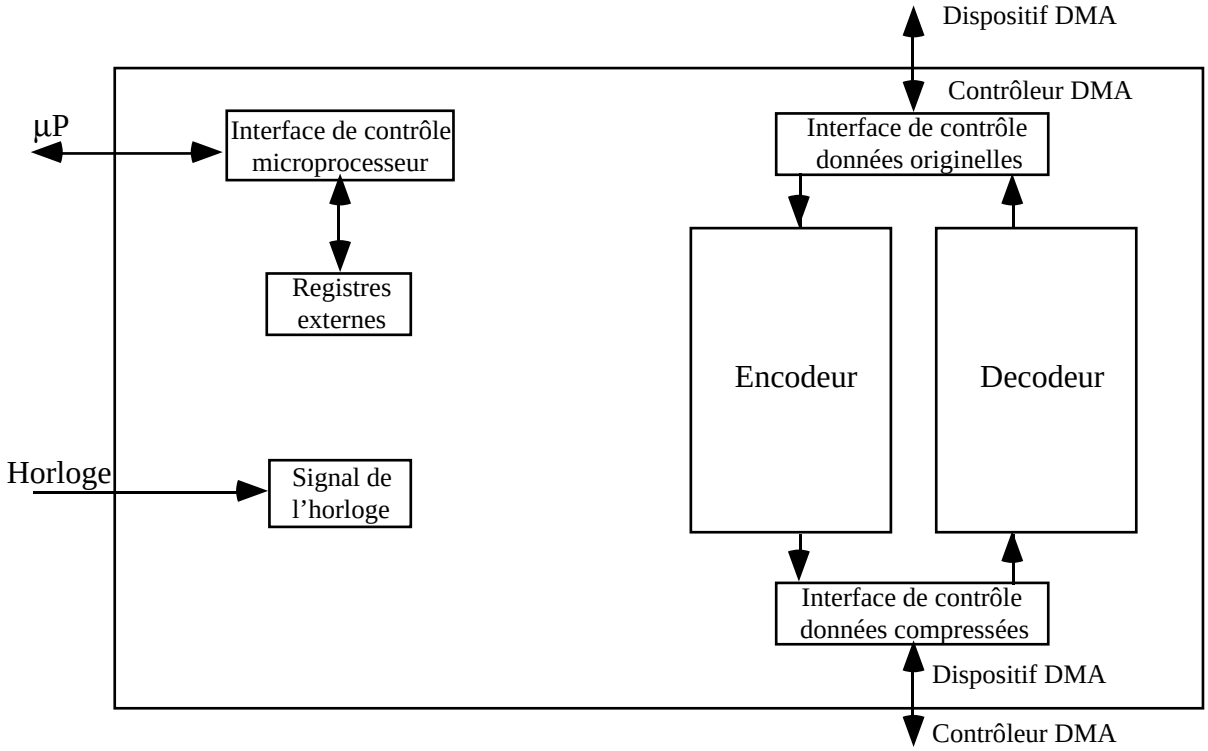


Figure 5.10 Circuit de compression et décompression de données.

Ce type de circuits utilisent des algorithmes de compression et décompression basés sur la méthode dite de Ziv et Lempel [ZL77]. Une caractéristique importante de ces algorithmes est qu'ils peuvent effectuer la compression et la décompression des données au fur et à mesure que celles-ci sont transmises à travers le réseau. En effet, d'une part le dictionnaire de relations entre les chaînes originales et leurs codes associés est généré pendant la compression au fur et à mesure que les données sont lues. D'autre part, ce dictionnaire peut être régénéré au fur et à mesure que les données sont décompressées.

Compression et décompression des données peuvent alors chevaucher leur exécution avec la transmission des données (exécution en pipe-line). Si l'on néglige (et pour le transfert de gros messages nous pouvons le faire) le retard causé par la compression et la décompression des premiers octets du message, il suffit que les débits de compression et de décompression soient supérieurs au débit de transfert pour que la compression de messages soit efficace comme méthode pour améliorer les performances d'un système d'acheminement de messages sur un réseau d'interconnexion.

La figure 5.11 montre l'intégration des circuits de compression et de décompression dans un réseau de communication point-à-point. La compression est contrôlée par le processeur de calcul qui, en fonction de la taille du message à envoyer, décide d'activer ou de désactiver le

circuit de compression. A l'arrivé d'un message l'en-tête de celui-ci permet au circuit de décompression de déterminer s'il y a une décompression à effectuer.

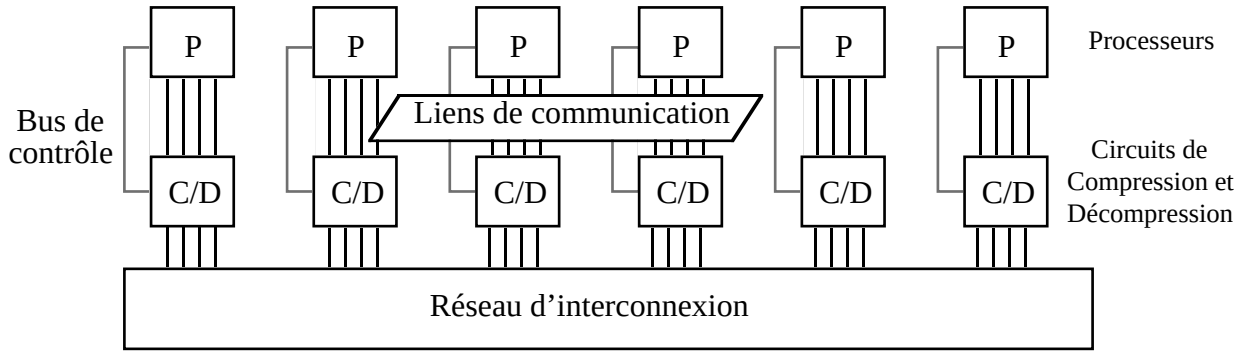


Figure 5.11 Intégration des circuits de compression et décompression dans un réseau d'interconnexion point-à-point.

Dans une architecture comme celle de la figure 5.11, des circuits de compression économiques satisfont les conditions requises pour garantir l'efficacité du système. En effet, si nous prenons par exemple un réseau de processeurs T9000 [INMOS91], où le routage est assuré par des processeurs spécialisés C104, le débit de transmission de pointe est de 12 Méga-octets/s avec une moyenne effective inférieure à 2 Méga-octets/s selon le réseau. Aujourd'hui il existe sur le marché des circuits de compression et décompression avec un débit de 5 Méga-octets/s à moins de $20 l'unité.

5.4 Conclusion

L'architecture actuelle des SF est une conséquence de l'organisation moderne des systèmes d'exploitation. Afin de faciliter la tâche de conception et de mise en œuvre de systèmes d'exploitation ouverts, dans un environnement où la technologie évolue continuellement, les développeurs des systèmes d'exploitation ont orienté leurs solutions vers une approche micro-noyau. Cette approche place le SF au niveau utilisateur en tant que serveur de données et laisse le contrôle de l'interaction inter-processus à la charge du micro-noyau.

Pour qu'une solution réussisse dans un environnement massivement parallèle, elle doit prendre en compte tous les niveaux architecturaux du système. L'architecture logicielle des E/S concerne alors non seulement le SF mais aussi tout le système d'exploitation et particulièrement son micro-noyau. De ce fait, cette deuxième partie du rapport est consacrée aux aspects du système d'exploitation qui peuvent aider à la conception d'une couche logicielle des E/S en rapport avec les diverses demandes des applications.

Dans ce chapitre nous avons introduit ParX, un micro-noyau générique adéquat pour les architectures massivement parallèles. Ce noyau est structuré comme un ensemble de machines virtuelles qui permet aux utilisateurs (et en particulier aux développeurs système) de définir le niveau d'abstraction désiré et d'éviter ainsi les surcoûts induits par des services dont ils n'ont pas besoin.

L'analyse du système ParX nous a amené à lui apporter quelques modifications pour mieux l'adapter aux besoins spécifiques des sous-systèmes d'E/S. Nous avons proposé trois extensions à ParX, chacune à un niveau différent du micro-noyau. D'abord, l'extension du modèle de processus pour dégager l'entité système à laquelle limiter la cohérence offerte pour les accès concurrents aux fichiers. Nous avons ainsi proposé la ligue, entité qui regroupe plusieurs Ptâches qui peuvent coopérer par partage de fichiers, ceci grâce à la cohérence de l'information garantie par le système. Ensuite, nous avons défini le cluster mémoire qui sert de mémoire auxiliaire aux applications, ce qui établit une hiérarchie de mémoires qui limite les accès aux dispositifs de stockage externe. Enfin, nous avons présenté un protocole pour accélérer le transfert de gros messages (cas typique des opérations d'E/S) basé sur l'utilisation de chemins alternatifs et la compression de données. Bien que la compression ne soit pas envisageable pour être réalisée au niveau du micro-noyau, nous avons montré l'utilité de ce mécanisme et la faisabilité de son intégration, par des circuits matériels, aux architectures actuelles.

Chapitre 6
La mémoire virtuellement partagée

La mémoire virtuelle est un service du niveau système qui peut être utilisé pour faciliter l'exploitation efficace des sous-systèmes d'E/S actuels. La projection des fichiers dans l'espace d'adressage virtuel (*mapped file I/O*) permet aux applications d'utiliser les mécanismes de mémoire virtuelle pour accéder aux fichiers. La mémoire vive peut être utilisée ainsi comme un cache de la mémoire secondaire, ce qui induit un surcoût inférieur à la méthode traditionnelle des tampons système [KSU93], et qui décharge le système de fichiers, sinon de tout, au moins d'une grande partie de la gestion de la cohérence des données [Ca91].

Or, si dans les architectures à mémoire partagée la mémoire virtuelle n'est pas une technique trop pénalisante, elle est beaucoup plus complexe dans une architecture où chaque processeur manipule un espace d'adressage indépendant. Le principal but d'une mémoire virtuelle est alors d'offrir un espace d'adressage virtuel commun à tous les processus qui s'exécutent sur les différents processeurs de la machine. Il s'agit ici de doter le système d'exploitation d'un mécanisme pour accéder de façon transparente à l'ensemble des mémoires locales, et d'un protocole de cohérence pour la gestion des accès concurrents à ces mémoires.

Ce chapitre décrit les mécanismes de base nécessaires à l'implémentation d'un espace d'adressage commun dans une architecture extensible à mémoire distribuée, et leur réalisation dans PAROS. Mais avant d'exposer notre solution nous présentons la problématique de la mémoire virtuelle dans les architectures sans mémoire commune et l'approche prise dans PAROS pour y apporter une solution.

6.1 La mémoire partagée dans les architectures sans mémoire commune

Les systèmes parallèles sans mémoire commune fondent leur modèle de programmation sur l'échange de messages [Ho85]. Bien qu'un tel modèle corresponde tout à fait aux caractéristiques de ces architectures, la programmation de processus qui ne peuvent communiquer que par échange de messages est très éloignée du modèle de programmation à variables partagées plus naturel aux utilisateurs [LKBT92].

Alors que la simulation du modèle à échange de messages est facilement réalisable sur une architecture à mémoire partagée, le contraire est beaucoup plus complexe. Les systèmes à mémoire virtuellement partagée (*Virtual Shared Memory*) ont été conçus pour remédier à ce problème. Cette approche implémente au-dessus du système d'échange de messages, un espace d'adressage unique pour tous les processeurs de la machine. Les systèmes IVY de l'université de Princeton [Li88], Munin de l'université Rice à Houston [BCZ91] et KOAN du laboratoire IRISA à l'université de Rennes [LP92] en sont des exemples de réalisation ; une comparaison des algorithmes utilisés pour l'implémentation d'une mémoire partagée à partir des mémoires privées peut être trouvé dans [SZ90]. Malgré le surcoût nécessairement introduit par cette solution, les systèmes à mémoire virtuellement partagée sont nécessaires pour faciliter la programmation et le portage de certaines applications dans les architectures massivement parallèles.

La demande croissante de la part des utilisateurs pour pouvoir utiliser le modèle de programmation à variables partagées sur des machines à mémoire distribuée, et les faibles performances affichées par les implémentations logicielles des mémoires virtuellement partagées, a amené certains constructeurs à intégrer dans leurs architectures les fonctionnalités nécessaires pour assurer le support matériel de ce modèle, comme par exemple la machine KSR de Kendall Square Research [KSR92] et la machine DASH de l'université de Stanford [LLGW92]. Récemment, le projet FLASH de l'université de Stanford a proposé une architecture COMA (**C**ache **O**nly **M**emory **A**rchitecture ou Architecture mémoire basée strictement sur des caches) qui est une architecture extensible avec une mémoire partagée composée d'un ensemble de mémoires distribuées. Ici, le concept de base est l'extension de la notion de cache à l'intégralité des différentes mémoires locales du système. La mémoire de chaque nœud (composé d'un ou plusieurs processeurs et d'une mémoire locale) est ainsi considérée comme un grand cache pour les processeurs du nœud, et les données sont automatiquement déplacées ou répliquées dans les mémoires des nœuds dans lesquels elles sont référencées [HLH92, KOHH94].

Le paradigme fondamental considéré par ParX est celui de l'échange de messages car dès sa conception, PAROS a été prévu pour s'exécuter aussi bien sur des machines à mémoire commune que sur des architectures à mémoire distribuée. La raison d'être des machines virtuelles de ParX est d'offrir l'abstraction la mieux adaptée aux besoins de chaque utilisateur du système ; il est donc naturel, que parmi les diverses machines virtuelles proposées, il en existe une qui fournisse aux utilisateurs un espace d'adressage unique et cohérent qui supporte le modèle de programmation à mémoire partagée. C'est l'objectif de la machine virtuelle à mémoire virtuellement partagée DIVA (**DI**stributed **V**irtual memory **A**pproach) [BCDE93, BCM95].

6.2 Architecture de DIVA

DIVA est un gestionnaire de mémoire virtuellement partagée spécialement conçu pour les architectures massivement parallèles. DIVA est un serveur distribué qui s'exécute sur les différents sites de la machine dans lequel un effort particulier a été porté pour éviter la centralisation des données nécessaires à son exécution. Un processus qui s'exécute sur n'importe quel site peut accéder à toutes les portions de l'espace virtuel défini par la machine virtuelle à mémoire partagée. Comme le montre la figure 6.1, le gestionnaire de mémoire effectue la correspondance entre les mémoires locales et l'espace d'adressage virtuel. De plus, il garantit la cohérence des données à l'intérieur de l'espace partagé [BM94].

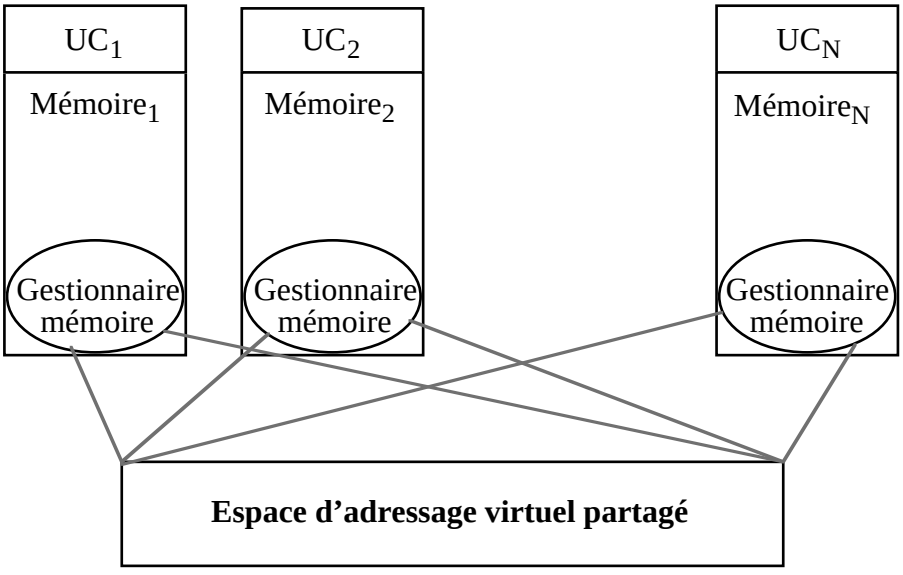


Figure 6.1 Serveur de mémoire virtuellement partagée.

L'espace d'adressage virtuel est divisé en pages de taille fixe et de ce fait, la page est l'unité d'accès et de cohérence. Toutefois, une couche supplémentaire est prévue pour manipuler des

objets de plus grosse granularité (p. ex. des segments). Le système ajuste les droits d'accès à chaque page (nul, lecture exclusive, etc.) de telle sorte qu'une violation des droits d'accès, qui peut affecter la cohérence du système, génère un défaut de page. Transparent à l'utilisateur, le mécanisme de gestion de défauts de page prend en charge toute violation d'accès.

La figure 6.2 montre la structure du serveur de mémoire virtuellement partagée qui s'exécute sur chaque nœud. Il est composé à sa base d'une unité matérielle de gestion de mémoire (MMU pour **M**emory **M**anagement **U**nit), d'un gestionnaire de mémoire et d'un démon scanner[1] qui sert à maintenir l'occupation de la mémoire entre deux seuils prédéfinis.

Figure 6.2 Structure interne du serveur mémoire.

L'unité de gestion de mémoire est normalement intégrée dans le matériel ; elle est responsable de la traduction automatique des adresses virtuelles en adresses réelles. Une référence à une adresse mémoire dans l'espace virtuel génère un défaut de page lorsque la page associée à l'adresse référencée ne se trouve pas dans la mémoire physique du nœud ou lorsque la copie de la page disponible localement ne dispose pas des droits d'accès demandés. La MMU fait alors appel au gestionnaire de mémoire pour qu'il retrouve la dite page à partir des disques ou des mémoires locales des autres nœuds. Les mémoires distantes sont donc un autre niveau dans la hiérarchie des mémoires entre une mémoire locale et les disques.

6.2.1 Le démon scanner

Le démon scanner est un processus qui s'exécute en arrière-plan, et dont le rôle est d'examiner l'état de la mémoire et de réévaluer les décisions de placement déjà prises. Le serveur de mémoire virtuellement partagée dispose de deux scanners : un pour la libération de pages de la mémoire locale du nœud et un autre pour le préchargement de données. L'activité de ces scanners est déterminée au démarrage du système, ils peuvent être activés (ou désactivés) séparément.

[1] De l'anglais Scanner Daemon

6.2.1.1 Scanner de libération de pages

Ce processus libère des pages mémoires lorsque la mémoire physique du nœud est surchargée. Le chargement d'une page lors de la résolution d'un défaut de page peut devenir très coûteux s'il doit être précédé de l'exécution d'un algorithme de remplacement de pages [Ba94]. Le but du scanner de libération de pages est alors de diminuer le risque d'avoir à exécuter cet algorithme de remplacement au moment d'un défaut de page.

6.2.1.2 Scanner de préchargement de pages

En fonction des références mémoire faites par l'application, ce processus est capable de précharger les pages candidates à être référencées dans un futur proche. L'objectif est donc de limiter le nombre de défauts de pages généré par des accès aux informations distantes. Les pages préchargées sont gelées en mémoire pendant un certain temps pour éviter qu'elles soient choisies par l'algorithme de remplacement.

La figure 6.3 montre les zones d'activité des deux scanners. Les pages mémoire libres sont organisées dans une liste chaînée, et un compteur global est maintenu pour déterminer à tout moment le nombre de pages libres dans la mémoire physique. La frontière d'activité de chaque scanner est déterminée par les seuils d'occupation de la mémoire.

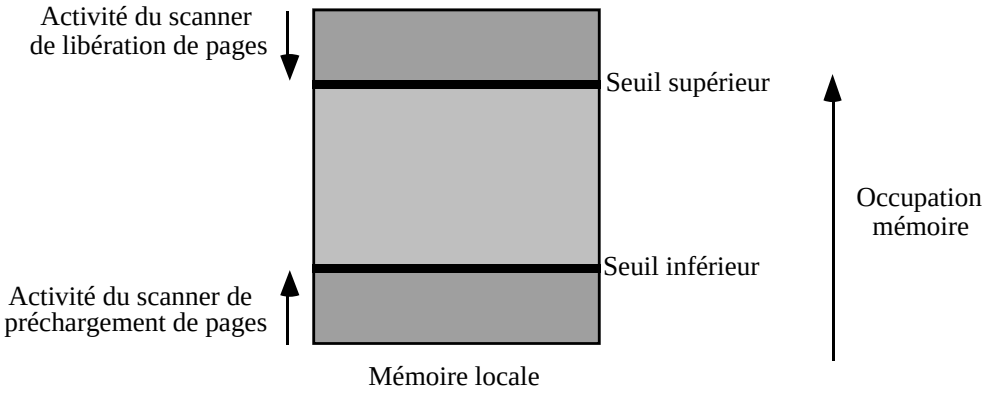


Figure 6.3 Zones d'activités des démons scanners.

6.2.2 Le gestionnaire de mémoire

Tout comme avec la mémoire virtuelle traditionnelle [Ta87, Kr88], la mémoire virtuellement partagée n'existe que *virtuellement*. Une référence à une adresse dans l'espace d'adressage non disponible dans la mémoire physique génère un défaut de page. Pour résoudre ce défaut de page, le gestionnaire de mémoire récupère la page pertinente du disque ou de la mémoire locale

d'un autre nœud et fait une copie dans la mémoire locale du nœud appelant. Suite à cette opération, s'il reste des copies de cette page dans la mémoire locale d'un autre nœud, le gestionnaire met en œuvre les mécanismes nécessaires pour garantir la cohérence des informations qui y sont stockées [Ca93].

6.3 Un espace d'adressage unique pour PAROS

Dans PAROS, les machines virtuelles sont le moyen qu'ont les utilisateurs pour définir le type de support d'exécution et l'ensemble de fonctionnalités requis par leurs applications. Les machines virtuelles s'appliquent sur le cluster associé aux applications lors de leur chargement, ce qui signifie que pour DIVA l'espace d'adressage commun est accessible à tout le cluster, c'est-à-dire, à tous les threads des différentes Ptâches qui s'exécutent dans ce cluster [BCDE93, Ca93]. Ce modèle reste cohérent avec notre conception de la ligue car il propose un mécanisme très efficace de coopération, la mémoire partagée, aux Ptâches d'une ligue.

La difficulté fondamentale à laquelle se heurte le gestionnaire de mémoire est la résolution d'un défaut de page. Le problème est ici plus complexe que dans les systèmes traditionnels à mémoire commune, car si dans ces systèmes la recherche d'une page ne se fait que de dans deux espaces de recherche (la mémoire vive et les disques), dans un système à mémoire distribuée, le nombre d'espaces de recherche est proportionnel au nombre de mémoires privées dans la machine.

Dans cette section nous proposons un mécanisme original pour retrouver une page dans l'espace virtuel d'un cluster et pour garantir la cohérence des informations à l'intérieur de cet espace. Pour cela, nous définissons tout d'abord l'espace de recherche des pages dans PAROS. La solution est extensible par rapport au nombre de nœuds et à la taille de la mémoire du cluster ; et afin de maximiser ses performances elle exploite directement la couche de routage de ParX [Ca91, Ca93].

6.3.1 Hiérarchie de la mémoire

Les clusters mémoires définis dans le chapitre précédent servent à délimiter l'espace de recherche d'une page de la mémoire virtuelle. Nous pouvons spécifier une hiérarchie de recherche pour l'imposer au gestionnaire de mémoire lors de la recherche d'une page. La figure

6.4 montre cette hiérarchie ; évidemment, plus on monte dans la hiérarchie de mémoires, plus il est coûteux d'y accéder, l'ordre de recherche d'une donnée est alors déterminé de façon à diminuer le coût global de l'opération. La décision de stocker une page mémoire à un niveau ou à un autre de la hiérarchie est donc capitale pour les performances du système.

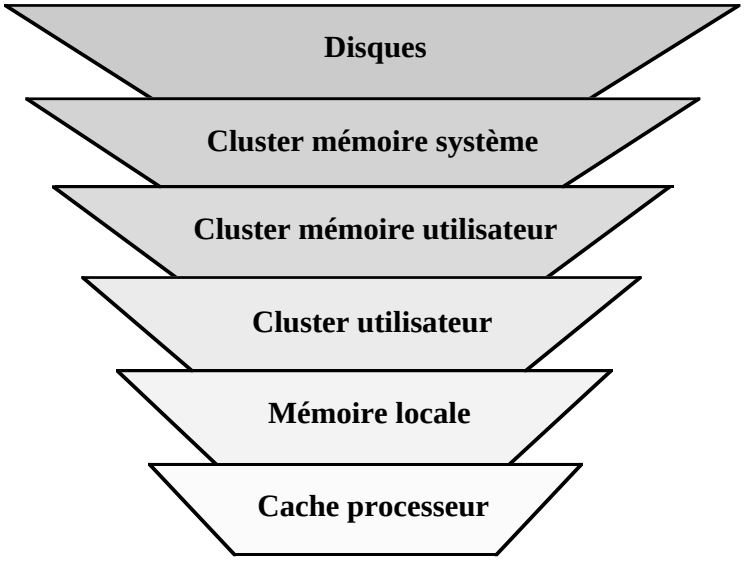


Figure 6.4 Hiérarchie de mémoires dans PAROS.

Niveau 0 : le cache du processeur géré directement par le matériel, de par sa petite taille (quelques kilo-octets) il ne contient pas de page entière mais les mots mémoires candidats à être référencés dans un futur proche.

Niveau 1: la mémoire privée du nœud. Si la page recherchée se trouve dans la mémoire locale du nœud, le système ne génère un défaut de page que si les droits d'accès ne correspondent pas au type d'accès demandé (p. ex. accès demandé en écriture sur une page en lecture exclusive).

Niveau 2 : le cluster utilisateur. Lors du chargement d'une ligue, le scanner de préchargement de pages distribue les pages nécessaires à l'exécution de la ligue sur tous les nœuds du cluster dans la limite de l'espace physique disponible. Ce préchargement initial de pages, fait du cluster utilisateur un très bon candidat pour avoir une copie de la page recherchée.

En outre, comme l'espace d'adressage commun est partagé par tous les processeurs du cluster utilisateur, et que les Ptâches qui s'exécutent dans ce cluster peuvent coopérer à travers cet espace global, il est probable que la page recherchée ait déjà été accédée sur un autre nœud et qu'elle se trouve alors dans la mémoire privée de ce nœud. La recherche d'une page à l'extérieur de la mémoire locale d'un nœud doit alors commencer par les mémoires privées des nœuds du cluster utilisateur.

Niveau 3 : le cluster mémoire utilisateur. Le cluster mémoire est associée à une ligue pour lui servir d'extension mémoire lorsque la mémoire physique du cluster ne suffit plus à stocker les données nécessaires à l'exécution de ses Ptâches. Un pont de communication direct entre le cluster de l'utilisateur et son cluster mémoire permet un accès rapide aux données qui y sont stockées. Le gestionnaire de mémoire utilise ce cluster comme cache mémoire pour toute la ligue, il est par conséquent le premier candidat pour satisfaire la recherche d'une page à l'extérieur du cluster utilisateur.

La recherche à l'intérieur du cluster mémoire utilisateur ne nécessite pas d'un parcours des processeurs à l'intérieur de ce cluster. En effet, comme toutes les demandes de chargement de pages vers le cluster système sont gérées par les processeurs de contrôle du cluster, ceux-ci peuvent conserver une table des pages stockées dans les mémoires locales des processeurs de ce cluster. De cette façon, la recherche d'une page à ce niveau est une opération qui n'implique que les processeurs de contrôle et le processeur qui dispose d'une copie de la page le cas échéant.

Niveau 4 : le cluster mémoire système. Une fois que tous les niveaux de mémoire associés directement à une ligue sont épuisés, le gestionnaire mémoire de la ligue en conclut que la page virtuelle recherchée n'est pas chargée dans l'espace physique alloué à l'utilisateur, et demande son chargement à la ligue système. Le gestionnaire de mémoire calcule alors le(s) bloc(s) physique(s) disque où la page virtuelle se trouve et commande l'opération d'E/S correspondante à la ligue système (Ptâche de gestion des périphériques).

Le cluster mémoire système représente les tampons système où les blocs disque sont stockés temporairement afin de limiter les accès aux unités externes (modèle traditionnel des tampons système [Ba86]). La ligue système pourrait alors retrouver le(s) bloc(s) demandé(s) par le gestionnaire mémoire de l'utilisateur dans ce cluster. Si dans le cluster mémoire système se trouvent les informations nécessaires pour résoudre le défaut de page, la Ptâche mémoire système envoie les données directement au cluster utilisateur (sans passer par le cluster système) à travers le cluster de contrôle (cf. figure 5.9).

Niveau 5 : les disques. Comme dans les systèmes traditionnels, la Ptâche de gestion des périphériques commande l'opération d'E/S et installe le(s) bloc(s) lu(s) dans le cluster mémoire système, d'où les données sont envoyées au cluster utilisateur comme nous l'avons déjà décrit.

Une fois la page est retrouvée, elle est copiée dans la mémoire locale du processeur où le défaut de page a été généré, ce qui permet aux accès ultérieurs à cette page, dans le même nœud, d'être locaux. Ceci nous est dicté par le principe de *localité de référence* : sur une période assez brève d'observation, la répartition des références à la mémoire présente une certaine stabilité ; autrement dit, à un instant donné, les références observées dans un passé récent sont en général une bonne estimation des prochaines références [De72]. Une des principales justifications de la mémoire virtuelle est le haut degré de localité des programmes séquentiels. Bien que les références dans les programmes parallèles puissent avoir un comportement différent du point de vue de la localité, chaque thread à l'intérieur d'une Ptâche exécute un code purement séquentiel et de ce fait, il respecte en général ce principe de localité.

La copie des pages de l'espace d'adressage virtuel dans les mémoires locales des processeurs introduit le problème de cohérence : la même page virtuelle se trouve physiquement dans des mémoires indépendantes ce qui peut entraîner des évolutions différentes. Un protocole de maintient de cohérence doit être alors mis en place afin d'assurer l'exécution (et la coopération) correcte des Ptâches à l'intérieur d'un cluster à mémoire virtuellement partagée.

PAROS a été développé dans le souci de couvrir un large type d'applications, ce qui implique que ses mécanismes de base ne doivent pas favoriser un type de comportement aux dépens d'un

autre. La gestion de la cohérence est en conséquence flexible et le modèle à assurer est un choix laissé à l'utilisateur [BM94]. Nous considérons que le modèle de cohérence qui doit être supporté par les gestionnaires de mémoire dépend de la façon dont les applications accèdent aux variables partagées. Contrairement au système Munin [BCZ91], qui demande à l'utilisateur de spécifier pour chaque variable le protocole de cohérence qui doit être utilisé, afin de faciliter l'utilisation de la machine virtuelle, nous préférons plutôt de lui demander de définir le modèle global de cohérence qui répond le mieux aux besoins de son application.

De même la gestion des E/S nécessite un modèle suffisamment flexible pour prendre en compte les divers types d'accès que font les applications aux fichiers. Pour répondre à cette exigence, notre implémentation combine les modèles de cohérence forte et de cohérence faible [Mo93]. A cet effet, nous utilisons trois types d'accès pour une page : WEAK, READ et WRITE (faible, lecture et écriture). Le premier nous permet d'accéder à une page mémoire même si celle-ci est en train d'être modifiée sur un autre nœud (accès aux fichiers sans contrôle de cohérence), et les deux derniers garantissent que les données lues dans une page soient effectivement celles qui ont été écrites en dernier (définition de cohérence forte). Chaque type d'accès impose certaines restrictions pour le partage des copies des pages :

WEAK : c'est une version qui peut ne pas être à jour. La modification d'une page en accès WEAK ne concerne que le nœud où se trouve cette page. Un nœud demande une page WEAK lorsqu'il ne lui est pas nécessaire d'obtenir la toute dernière version de la page.

READ : c'est une copie avec la dernière version de la page. Il peut exister plusieurs copies de cette page avec le même type d'accès sur des nœuds différents du cluster ;

WRITE : La page peut être modifiée et il n'existe aucune autre copie de cette page avec l'accès READ ou WRITE dans tout le cluster y compris le cluster mémoire du cluster utilisateur. Les détenteurs de copies de la page en état WEAK ne sont pas prévenus de la modification.

6.3.2 Les gestionnaires de K. Li et P. Hudak

Dans [LH89], K. Li et P. Hudak proposent trois solutions pour retrouver une page dans un réseau de processeurs sans mémoire commune : à partir d'un serveur centralisé, d'un serveur distribué fixe, ou d'un serveur distribué dynamique.

La gestion centralisée des pages consiste à définir un nœud unique du réseau comme le serveur de pages. Ce serveur connaît le contenu de toutes les mémoires locales des autres nœuds et de ce fait peut répondre à toute requête de recherche d'une page. Il est évident que cette solution ne peut être retenue que pour des réseaux où le nombre de nœuds est fortement limité.

La gestion distribuée fixe répartit la charge de gestion des pages entre tous les nœuds du réseau. Le système attribue à chaque nœud un ensemble de pages dont le nœud doit assurer complètement la gestion ; on dit alors qu'un nœud est le *propriétaire* de toutes les pages de son ensemble. Les pages ne changent jamais de propriétaire, et la correspondance entre les pages et les nœuds se fait à partir d'une fonction de hachage. L'écriture d'une page est une attribution exclusive du propriétaire de la page et tous les autres nœuds doivent s'adresser au propriétaire pour qu'il leur octroie le droit de modifier le contenu de la page. Cette solution est extensible par rapport au nombre de nœuds, malheureusement ses performances sont trop dépendantes de la distribution de l'ensemble de pages.

Dans la solution distribuée dynamique, la gestion d'une page est assignée en fonction de la demande. Le principe de cette approche est que c'est au nœud qui demande le plus de droits d'accès sur une page que revient tout naturellement la charge de sa gestion. Une page peut ainsi changer de propriétaire au milieu de l'exécution d'une application. Cette dynamicité rend plus complexe la recherche d'une page dans le réseau. Un système de propriétaires probables (*probOwner*) est utilisé pour garder la trace d'une page : à chaque changement de propriétaire, l'ancien gestionnaire garde un pointeur vers le nouveau, de façon à toujours disposer d'un moyen d'accès à partir du gestionnaire original (voir figure 6.5). La difficulté de trouver le placement idéal des pages dans le cas de la gestion distribuée fixe fait de la gestion distribuée dynamique une solution plus adaptée pour les systèmes universels ; toutefois, comme nous le verrons par la suite, l'espace mémoire pour représenter les structures nécessaires à son exécution ne permettent pas son implémentation dans le cas des architectures extensibles [Ca93].

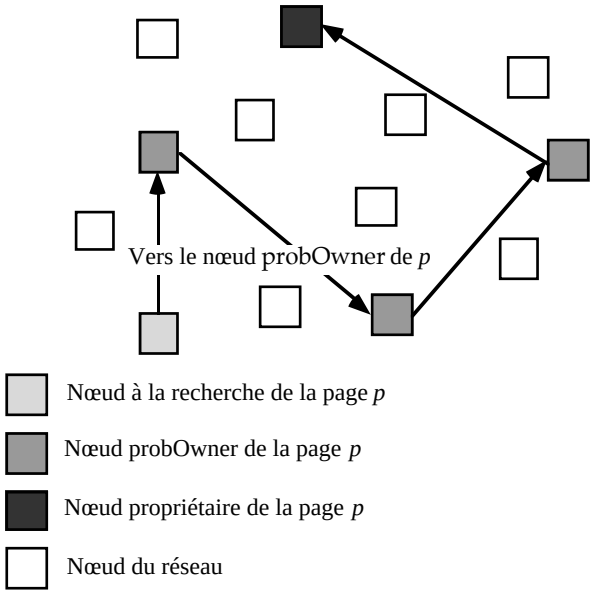


Figure 6.5 Liste de nœuds probOwner pour accéder à une page.

6.3.3 Un gestionnaire dynamique extensible

Pour la réalisation d'un gestionnaire dynamique extensible nous avons utilisé comme point de départ le principe de la gestion distribuée dynamique des pages, tout en conservant l'extensibilité de la gestion distribuée fixe. De plus, comme nous allons le montrer, nous avons tiré le meilleur profit des fonctionnalités du micro-noyau ParX pour maximiser l'efficacité de notre solution.

Pour retrouver une page dans un cluster, le gestionnaire de mémoire d'un nœud consulte dans un tableau local une indication sur la localisation de la page recherchée : une table *PTable* sert à cet effet et le contenu de `PTable[p].probOwner` est l'identification du propriétaire probable de la page *p*. Au moment de la distribution initiale des pages, les informations des champs probOwner sont exactes et le gestionnaire de la page *p* est effectivement celui désigné par `PTable[p].probOwner`. Cependant, une fois l'exécution des Ptâches commencée, les pages peuvent changer de gestionnaire et le champ probOwner n'est qu'un indice sur le véritable gestionnaire de la page.

Dans la version de Li et Hudak, la requête de recherche d'une page visite les nœuds spécifiés par la suite de pointeurs probOwner jusqu'à qu'elle parvienne au véritable propriétaire de la page. En effet, seulement le propriétaire d'une page peut répondre aux requêtes qui la concernent car il stocke dans sa mémoire privée l'identification de chacun des nœuds qui partage la page. Le changement de propriétaire d'une page n'est connu que par l'ancien propriétaire et par tous ceux qui disposaient d'une copie de la page au moment de l'application du modèle de cohérence associé à l'application. Ceci peut générer des listes de probOwner assez longues et ralentir ainsi les performances de l'algorithme de recherche d'une page. De plus, l'information des pointeurs probOwner ignore complètement la proximité physique possible sur le réseau entre une requête et un gestionnaire capable d'y répondre.

Nous faisons une différence importante entre une requête pour une page en lecture et une requête pour une page écriture. Alors qu'une requête d'écriture doit effectivement arriver jusqu'au propriétaire de la page pour mettre en œuvre le mécanisme de cohérence requis, les requêtes de lecture peuvent être résolues par n'importe quel nœud qui dispose d'une copie de la page. Ainsi, non seulement l'ensemble des pages est géré de façon distribuée, mais la gestion de chaque page est distribuée parmi tous ceux qui la partagent.

Une fois la requête d'écriture d'une page est parvenue au nœud propriétaire de la page, celui-ci implémente un protocole d'invalidation de toutes les copies de la page dont le type d'accès est READ. Après cette invalidation, le gestionnaire mémoire met à jour le champ probOwner pour cette page avec l'identification du nœud à l'origine de la requête. Ce nœud devient alors le nouveau propriétaire de la page.

La requête d'une page en lecture est également envoyée vers le nœud propriétaire, mais elle peut être résolue avant son arrivé à ce nœud. En effet, si sur le chemin vers le nœud propriétaire (en suivant la liste des nœuds probOwner) la requête visite un nœud qui dispose d'une copie de la page demandée, le gestionnaire de ce nœud peut arrêter la requête et y répondre. Pour qu'un gestionnaire puisse répondre à une requête, il faut tout de même qu'il dispose d'une copie avec un type d'accès compatible avec celui demandé. Ainsi, pour résoudre une requête d'accès READ il faut que le nœud dispose d'une copie de la page avec un accès READ. En revanche pour une requête d'accès WEAK, n'importe quel nœud avec une copie de la page peut répondre à la requête.

Le chemin vers le propriétaire d'une page suit la liste de nœuds probOwner qui a été créée au fur et à mesure que la page a changé de propriétaire. Une requête de recherche est alors transformée en une suite de messages point-à-point entre les membres de cette liste. Par ailleurs, les nœuds sur ce chemin sont ceux qui à un moment de l'exécution ont référencé la page recherchée et qui, de ce fait, ont eu une copie de la page. Ils sont en général des bons candidats pour avoir une copie de la page et donc pouvoir résoudre la requête.

L'acheminement des messages est assuré par la couche de routage du micro-noyau ParX : un message, pour être délivré, doit traverser un certain nombre de nœuds du réseau. L'information du probOwner n'est qu'un pointeur qui sert à garder la trace des changements de propriétaire subis par une page. Il n'y a aucune relation entre cette trace et la topologie physique du réseau. Or, la topologie du réseau est fondamentale dans les architectures parallèles et nous devons la prendre en compte au moment d'effectuer une recherche.

L'importance du réseau dans la recherche d'une page est montrée dans la figure 6.6. Le réseau physique est ici configuré en chaîne, chaque nœud a deux voisins, sauf le premier et le dernier qui n'en ont qu'un seul. Le numéro à l'intérieur de chaque nœud correspond à son identification dans le réseau. Si nous supposons que la page *p*, dont le propriétaire original était le nœud 1, a été référencée exclusivement en écriture par les nœuds 9, 3, 6 et 5 dans cet ordre, la valeur de `PTable[p].probOwner` locale à chaque nœud est celle notée en bas des nœuds. De plus, seul le nœud propriétaire de la page possède une copie de *p*. La figure montre la suite de messages nécessaires lorsqu'une recherche de la page *p* est lancée dans le nœud 1 : la requête suit exactement la même trajectoire que la page sans considérer l'interconnexion physique des nœuds. Dans cet exemple, la requête de recherche traverse 19 nœuds avant d'arriver au propriétaire de la page alors que dans le réseau il n'y a que 9 nœuds au total !

Notre recherche d'une page ne se limite alors pas aux nœuds désignés par les champs probOwner, mais elle fait aussi participer les nœuds qui seront traversés par la requête dans son acheminement par la couche routage de ParX. Dans l'implémentation actuelle de ParX les

messages sont routés au fur et à mesure qu'ils traversent le réseau : lorsqu'un nœud reçoit un message il vérifie si le message est arrivé à destination, et si ce n'est pas le cas il le renvoie par un lien de communication qu'il calcule selon la fonction de routage.

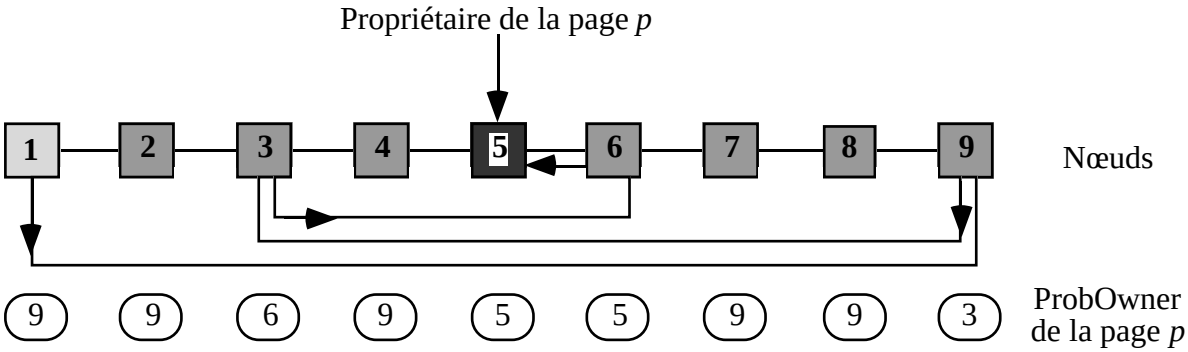


Figure 6.6 Recherche de la page *p* dans un réseau configuré en chaîne.

Pour exploiter le routage dans la recherche d'une page, il suffit alors de vérifier si la page recherchée est disponible dans la mémoire locale du nœud avant d'essayer de router le message (cf. figure 6.7). Pour l'exemple de la figure 6.6, la recherche de la page est résolue lorsque le message du nœud 1 vers le nœud 9 atteint le nœud 5, c'est-à-dire, la requête ne visite que 4 nœuds au total.

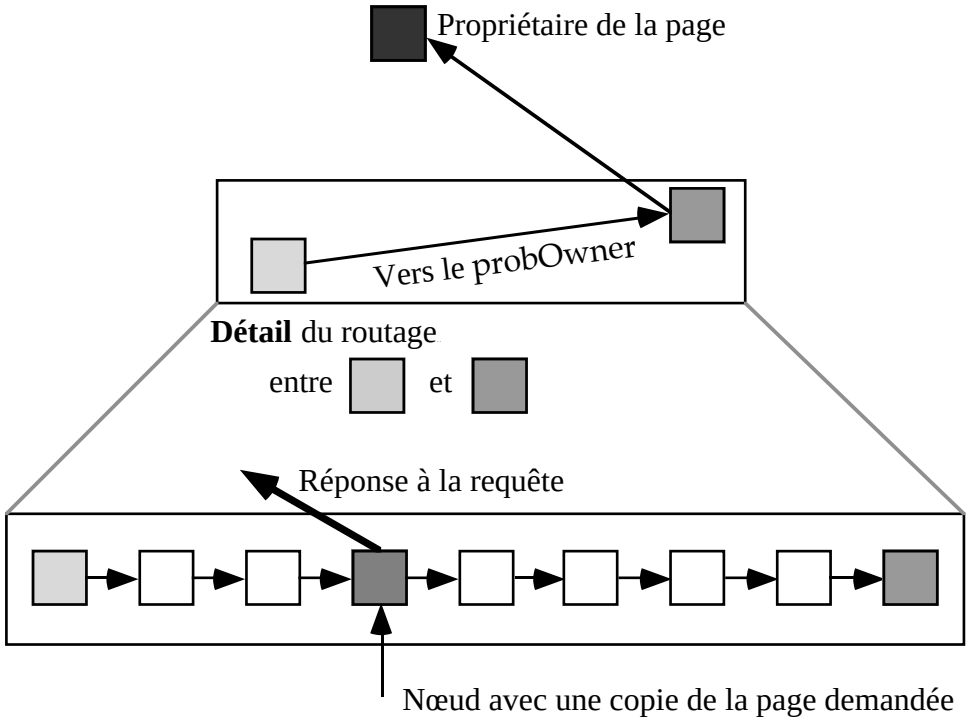


Figure 6.7 Recherche d'une page pendant le routage.

Même si la page recherchée n'est pas présente dans la mémoire locale d'un nœud, le surcoût[1] investi pour sa recherche pendant le routage n'est pas pour autant perdu, car nous pouvons profiter du passage sur un nœud pour améliorer l'information de celui-ci sur la localisation de la page. En effet, parmi les informations transmises dans la requête, se trouve l'identification du nœud qui demande la page. Ce nœud ne deviendra pas forcement le propriétaire de la page (notamment pour un accès en lecture), mais nous savons qu'il sera "bientôt" en possession d'une copie de la page. Si la page n'est pas dans la mémoire locale d'un nœud intermédiaire dans le routage d'une requête, le gestionnaire modifie la valeur du champ probOwner pour la page recherchée avec l'identification du nœud qui est à l'origine de la requête, ceci toutefois si la requête n'est pas pour un accès de type WEAK. Cette modification améliore la connaissance globale des nœuds sur la localisation des pages de l'espace virtuel.

De cette façon les requêtes ultérieures à cette même page émanant de ces nœuds intermédiaires seront dirigées vers le nœud qui a demandé la page. Il est évident, que ce nœud est celui qui est le plus probable pour avoir une copie de la page afin de répondre aux requêtes de recherche. Pour les requêtes d'accès WEAK et READ, ce nœud peut envoyer directement la page comme réponse à une requête, et pour les requêtes d'accès WRITE, il sera en mesure de renvoyer la requête vers le véritable propriétaire de la page.

La modification de la valeur du probOwner sur les processeurs intermédiaires est cohérente avec le rôle "d'indice" représenté par cette information. En effet, le nœud qui demande une page avec un accès READ ne deviendra pas le propriétaire de la page, mais il en obtiendra une copie à jour. Par ailleurs, il est le dernier nœud à avoir reçu une copie et il connaît l'identification du nœud propriétaire de la page comme nous le montre le théorème suivant.

Théorème 6.1 : *Tout nœud avec un accès READ sur une page connaît l'identification exacte du propriétaire de la page.*

Preuve : A la distribution initiale, seulement le nœud propriétaire de la page dispose d'une copie et donc seulement lui peut résoudre une requête pour cette page. Si un nœud demande l'accès READ sur une page, tous les nœuds qui participeront au routage de cette demande modifieront leur valeur de probOwner pour cette page avec l'identification du nœud qui fait la demande. Cette modification fait de ce nœud, vis-à-vis des autres, un nœud capable de répondre à des requêtes concernant cette page. De plus, comme il est le premier à faire une telle demande, il ne peut recevoir la copie de la page que du nœud propriétaire de la page, il connaît alors son identification. Une deuxième requête pour la même page pourra être résolue, soit par le propriétaire, soit par le nœud ayant reçu une copie. Dans les deux cas, l'identification du

[1] Dans la section 5.3.5.2 nous montrons que le surcoût sur chaque nœud est en fait une fonction de la probabilité de que la page se trouve dans le nœud.

propriétaire de la page peut être transmise. De cette façon à chaque fois qu'un nœud obtient une copie d'une page, il obtient aussi l'identification du nœud propriétaire. Par ailleurs, une page ne peut changer de propriétaire que si toutes les copies en accès READ sont invalidées.◊

6.3.4 Le problème de cohérence

Le problème de cohérence se manifeste lorsqu'une page de l'espace virtuel est répliquée sur plusieurs nœuds de la machine. Un défaut de page en lecture sur un nœud provoque une requête de recherche de la page dans le réseau, et une fois que l'algorithme de recherche trouve une copie avec un type d'accès compatible avec celui demandé, il la copie dans la mémoire locale du nœud qui a généré la requête. Des mécanismes de gestion de cohérence doivent être mis en place afin d'assurer que toutes les versions d'une page virtuelle respectent le modèle de cohérence spécifié par l'utilisateur qui a défini la machine virtuelle. Dans cette section nous présentons les mécanismes développés pour assurer que toutes les versions d'une page en accès READ sont identiques et correspondent à la dernière écriture effectuée sur la page.

La solution adoptée est l'invalidation des pages en lecture avant d'octroyer le droit en écriture : lorsqu'un nœud demande une page en écriture, toutes les autres copies dans le réseau avec un accès READ sont invalidées. Pour ce faire, le nœud qui reçoit la page envoie un message d'invalidation à tous les nœuds avec une copie de la page en accès READ. Seulement après confirmation que toutes les copies ont été invalidées, l'accès à la page est autorisé et ce nœud devient le nouveau propriétaire de la page.

L'invalidation d'une page dans un nœud consiste à changer son droit d'accès de READ à WEAK. Toutes les informations connues par ce nœud sur la page sont ignorées, sauf le champ probOwner. En effet, avec le message d'invalidation, un nœud reçoit l'identification du nouveau propriétaire de la page. Ainsi, si le nœud veut récupérer à nouveau l'accès READ à la page, il dispose de l'information la plus récente sur la localisation de la page.

Pour envoyer les messages d'invalidation le propriétaire d'une page doit connaître l'identification de tous les nœuds qui possèdent une copie de la page. Plusieurs solutions ont été proposées pour adresser ce problème mais leur utilisation est limitée dans le contexte des machines massivement parallèles : d'une part les solutions basées sur des répertoires complets comme les tableaux de bits complets sont très gourmandes en mémoire, d'autre part les répertoires incomplets comme les tableaux de bits limités nécessitent des mécanismes de diffusion dont le surcoût (temps d'exécution et surcharge du réseau d'interconnexion) est très important [GWM90, CFKA90].

La gestion de la cohérence a été soigneusement conçue pour pouvoir être intégrée au mécanisme de recherche des pages. Notre système de recherche d'une page ne prévient pas le propriétaire d'une page à chaque fois qu'une nouvelle copie est générée. Plus qu'extensible, la structure pour représenter l'ensemble de nœuds qui partagent une page est suffisamment flexible pour permettre sa mise à jour à partir des différents nœuds qui peuvent satisfaire une demande de recherche d'une page. Pour ce faire, l'ensemble des nœuds qui partagent une page est représenté sous forme d'une liste dispersée [JLGS90]. Ici, les nœuds eux-mêmes sont chaînés de façon à ce que tous les nœuds qui partagent une page forment une liste circulaire. Le nœud propriétaire d'une page n'a pas dans sa mémoire une liste d'identificateurs, mais un pointeur vers un nœud qui a une copie de la page. Ce dernier aura, lui aussi, un pointeur vers un autre nœud possesseur d'une copie de la page, et ainsi de suite jusqu'à contenir tout l'ensemble de nœuds.

Pour invalider toutes les copies d'une page il suffit alors d'envoyer un message à travers cette liste, et lorsque le dernier nœud invalide sa copie, il peut renvoyer un acquittement au propriétaire de la page. Le prix à payer par cette technique de représentation est la perte du parallélisme lors de l'invalidation des copies d'une page. En effet, dans les techniques classiques avec un répertoire pour stocker l'identification de tous les nœuds qui partagent une page, le propriétaire peut envoyer les messages d'invalidation, et recevoir les acquittements, en parallèle sur tous les nœuds.

La technique de listes chaînées n'est pourtant pas très pénalisante. Nous utilisons une liste circulaire doublement chaînée qui nous permet d'envoyer les messages d'invalidation dans les deux sens, et accélérer ainsi l'invalidation de toutes les copies. Par ailleurs, tandis que dans les solutions avec répertoire des nœuds qui partagent une page une invalidation nécessite l'envoie sur le réseau de deux messages par nœud (requête et acquittement), avec la représentation des listes seulement un message par nœud est nécessaire. En effet, un seul acquittement suffit pour prévenir le propriétaire : celui du premier nœud à recevoir deux requêtes d'invalidation. L'encombrement du réseau est alors inférieur dans notre solution.

La figure 6.8 montre notre stratégie de représentation pour l'ensemble des nœuds qui partagent une page. Dans la figure nous représentons le tableau PTable, avec les listes associées à certaines pages qui sont partagées par plusieurs nœuds. L'espace nécessaire à la représentation de tout l'ensemble des nœuds qui partagent une page est constant et égal à la taille des deux pointeurs. Un nœud peut se retrouver dans plusieurs listes chaînées, autant que de pages qu'il partage.

Lorsqu'un nœud obtient une copie d'une page avec l'accès READ, il doit être inclu dans la liste associée à la page. C'est le nœud qui répond à la requête qui se charge de cette tâche. Notre structure permet cette opération parce que l'ajout d'un nœud dans une liste ne concerne que les

nœuds entre lesquels le nouveau est rajouté. Le propriétaire de la page n'est donc pas informé de l'existence de nouveaux nœuds dans la liste.

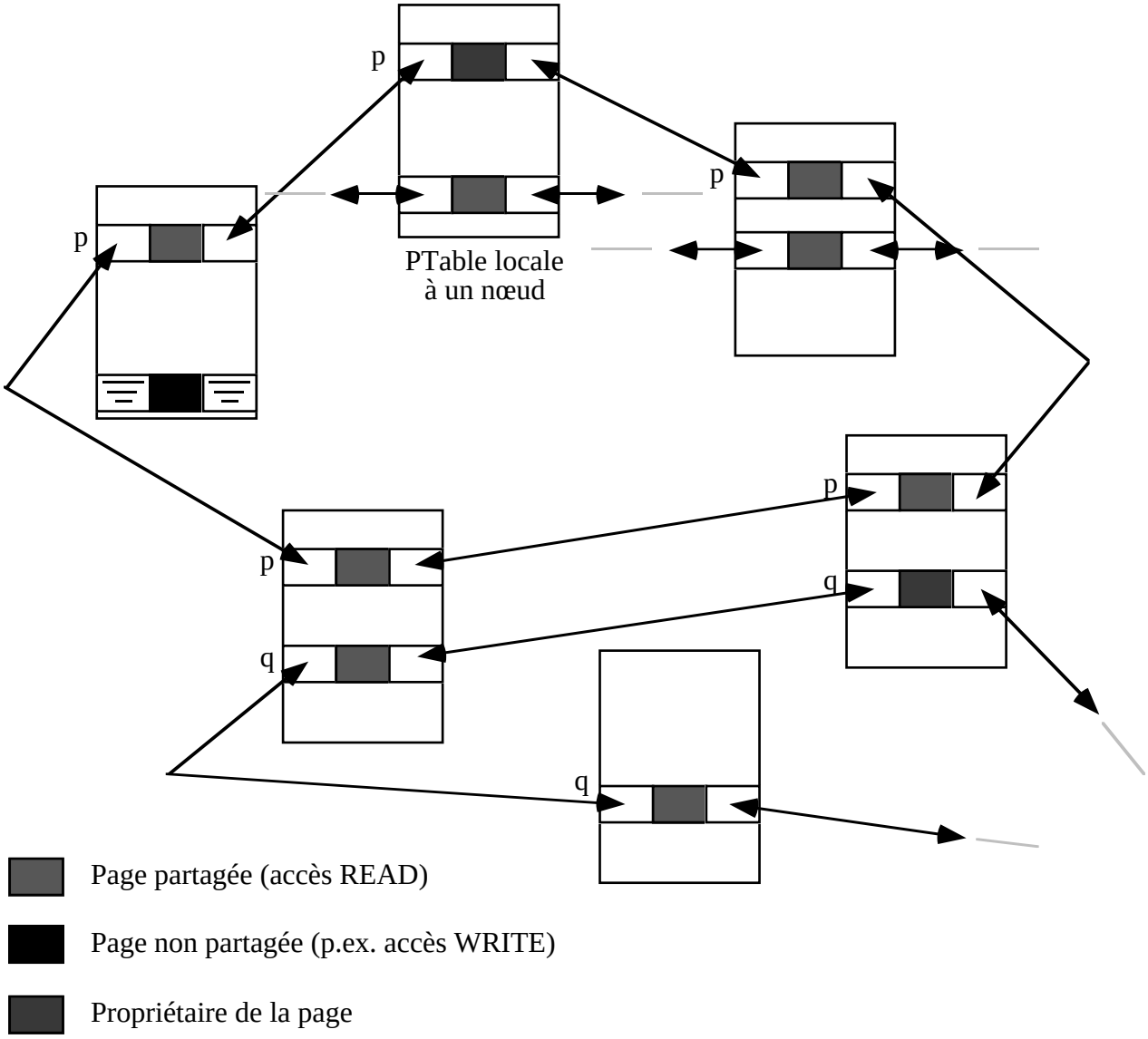


Figure 6.8 Représentation de l'ensemble de nœuds qui partagent des pages.

6.3.5 La table PTable

Le point faible de la distribution dynamique est l'espace mémoire nécessaire pour assurer la gestion de l'ensemble des pages de la mémoire virtuelle. En effet, la possibilité pour une page de changer de propriétaire dans le temps, oblige chaque gestionnaire à garder une trace (le pointeur probOwner) de sa localisation. La gestion distribuée dynamique utilise ainsi une table PTable avec une entrée par page dans l'espace virtuel.

L'espace mémoire occupé par PTable est alors proportionnel à la taille de l'espace virtuel. Il dépend aussi de la taille définie pour une page et des informations stockées pour chacune des pages (probOwner, pointeurs de la liste doublement chaînée, type d'accès, etc.). Nous pouvons calculer la taille de ce tableau :

Soient

b tel que 2^b est la taille de l'espace d'adressage utilisé
p tel que 2^p est la taille en octets d'une page
e tel que 2^e est la taille en octets de chaque entrée du tableau

Nous avons alors

$$taille(PTable) = \frac{2^b}{2^p} \times 2^e = 2^{b+e-p} \qquad (1)$$

De ce résultat, le seul paramètre que nous pouvons modifier à notre gré est la taille d'une page. La taille de l'espace d'adressage utilisé dépend des Ptâches à l'intérieur du cluster et est limitée par l'architecture matérielle ; et la taille de chaque entrée du tableau est limitée par les informations qui y sont stockées. De la formule (1) nous pouvons alors retenir que pour réduire la taille du tableau, il est nécessaire d'augmenter la taille des pages.

Cependant, plus la taille d'une page est grande, plus le risque de conflit entre plusieurs nœuds pour accéder à une page est grand. Un nœud qui veut modifier la valeur d'une variable doit avoir l'accès d'écriture sur toute la page où se trouve la variable. Ainsi, un autre nœud ne pourra accéder en lecture à aucune autre variable dans la même page. Bien que les programmeurs et/ou le système d'allocation puissent essayer d'éviter le risque de conflit en plaçant les données accessibles séparément sur des pages différentes, le gaspillage de mémoire qui résulte de cette solution est trop important.

La taille d'une page doit concilier ces deux critères contradictoires. Il n'existe pas de taille de page idéale pour toutes les applications, elle demeure alors un paramètre du système qui doit évoluer avec le temps[1]. Dans tous les cas, la taille d'une page doit permettre de profiter des mécanismes matériels disponibles. Si le matériel dispose par exemple d'une MMU, la taille des pages doit être adaptée aux spécifications de ce matériel.

La taille de l'espace d'adressage utilisé est une conséquence des besoins des Ptâches qui s'exécutent à l'intérieur du cluster. Le nombre et la taille de ces Ptâches sont fonction de la taille du cluster. Plus le cluster dispose de nœuds, plus il peut exécuter de grosses Ptâches.

[1] D'ailleurs dans les systèmes à mémoire paginée, cette taille n'a pas cessé d'augmenter et aujourd'hui il est normal de trouver des tailles de page de 4 ou 8 kilo-octets.

Nous pouvons dire qu'en général il existe une relation entre la taille de l'espace d'adressage utilisé et la taille du cluster. Bien que le contraire ne soit pas forcement vrai, nous pouvons faire l'hypothèse que plus le nombre de nœuds est important dans un cluster, plus la taille de l'espace d'adressage nécessaire à l'exécution des Ptâches de ce cluster est grande.

Cette hypothèse crée une liaison entre la taille de PTable et le nombre de nœuds dans un cluster. Or, dans une architecture extensible, la mémoire utilisée dans un nœud pour la gestion d'un service ne peut pas être une fonction du nombre de nœuds de l'architecture. Une telle dépendance risque d'épuiser rapidement la mémoire privée d'un nœud. Prenons par exemple un espace d'adressage de 2 Giga-octets (b=31), une taille de page de 4 kilo-octets (p=12) et une taille par entrée de 32 octets (e=5) ; la taille de PTable sur chaque nœud est alors de 16 Méga-octets !

En outre, l'utilisation de la mémoire virtuelle comme cache des E/S et le développement des architectures à 64 bits risquent d'aggraver la situation. En effet, les applications aujourd'hui deviennent de plus en plus gourmandes en mémoire et la capacité d'adresser des espaces de taille supérieure à 4 Giga-octets est une possibilité qui commence à s'avérer utile. Par ailleurs, la lecture et l'écriture de fichiers par projection dans la mémoire augmente considérablement la consommation de l'espace d'adressage.

Nous proposons de limiter PTable à un nombre constant d'entrées. Pour cela nous identifions quatre conditions pour qu'une page p nécessite une entrée dans le tableau PTable local à un nœud :

- le nœud est propriétaire de p ;
- le nœud est le *propriétaire original* de p (p était une des pages attribuées au nœud lors de la distribution initiale) ;
- le nœud dispose actuellement de l'accès READ sur p ;
- p est une des dernières pages référencées par le nœud (le nœud risque d'y faire référence à nouveau dans un futur proche). Le nombre de ces entrées est un paramètre du système.

Hormis la deuxième condition, nous pouvons dire que la taille de PTable ne dépend que des accès faits par chaque nœud. Le système peut calculer cette taille en fonction de l'espace d'adressage attribué aux tâches qui s'exécutent sur un nœud (zones de code, de données, de pile et de tas). Et si nous considérons le nombre de pages attribuées au nœud lors de la distribution initiale, nous obtenons une taille de tableau qui est inversement proportionnelle au nombre de nœuds.

La réduction du nombre d'entrées dans PTable ne va pas sans perte d'information. Comment fait un nœud pour accéder à une page dont il n'a pas d'entrée dans sa PTable locale ? Sans l'information du probOwner pour une page, le gestionnaire mémoire d'un nœud ne sait pas à qui s'adresser pour retrouver la trace de la page. C'est le même problème de la représentation par des répertoires incomplets de l'ensemble des nœuds qui partagent une page. Nous n'avons pas fait appel à la diffusion pour résoudre le premier (nous avons proposé les listes chaînées), nous utilisons le propriétaire original pour éviter de le faire maintenant.

Le propriétaire original peut être calculé par tous les nœud du cluster grâce à la fonction de hachage qui sert précisément à la distribution initiale des pages. Dans notre algorithme la requête de recherche d'une page, dont il n'existe pas d'entrée dans le tableau PTable, est adressée au propriétaire original de la page. Il nous reste à garantir que ce nœud est capable de retrouver le propriétaire courant de la page pour lui transmettre la requête.

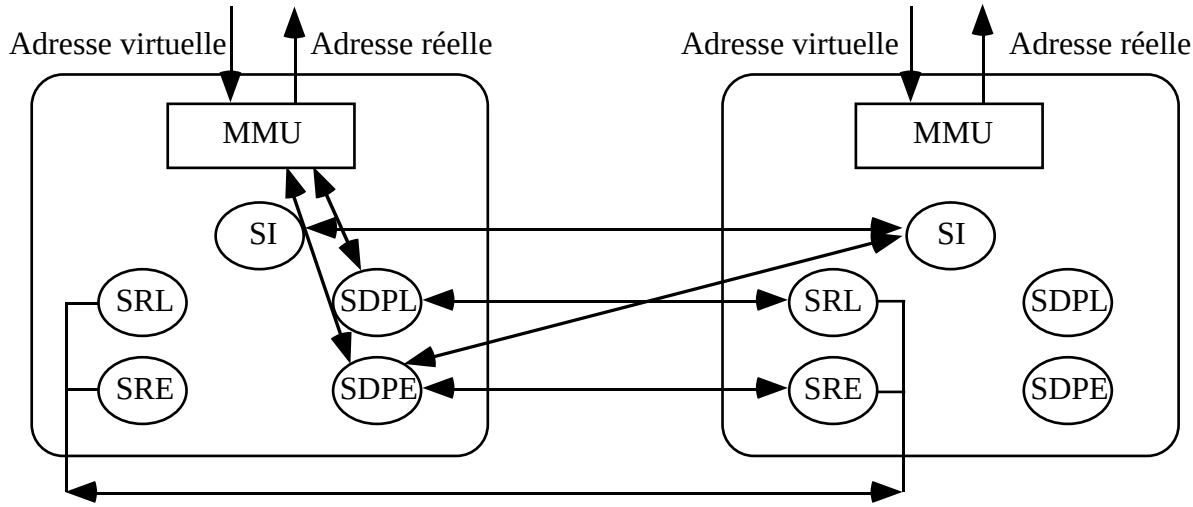
Pour cela, à chaque changement de propriétaire d'une page (en parallèle avec l'opération d'invalidation de toutes les copies en lecture), le nouveau propriétaire de la page prévient le nœud propriétaire original de ce changement. De cette façon, le nœud auquel on avait attribué une page lors de la distribution initiale connaît toujours la localisation de la dernière version de la page. Bien évidement, les modifications apportées au champ probOwner pendant le routage ne sont pas prises en compte par un nœud si elles concernent une des pages dont il est le propriétaire original.

6.3.6 Structure du gestionnaire de pages

Avec une taille bornée pour le tableau PTable nous pouvons réaliser un gestionnaire des pages d'une mémoire virtuelle partagée qui convient aux systèmes massivement parallèles. Comme le montre la figure 6.9, trois types de serveurs composent ce gestionnaire : les serveurs de traitement des défauts de pages, les serveurs de traitement des requêtes de recherche d'une page, et les serveurs d'invalidation de pages. Lorsque la MMU d'un nœud ne trouve pas la page correspondante à l'adresse virtuelle générée par le processeur, elle provoque un défaut de page qui est traité par un des serveurs de défauts de page. Celui-ci s'adresse au serveur de recherche de pages du nœud spécifié dans sa PTable locale, et reste en attente jusqu'à qu'une copie locale de la page est faite. Pour les cas d'une écriture le serveur de défaut de page en écriture envoie une requête au serveur d'invalidation avant de prévenir la MMU de l'installation de la page dans la mémoire locale. Nous détaillons dans les algorithmes de chacun de ces serveurs.

Chaque entrée du tableau PTable contient les informations suivantes pour chaque page :

`lock`	Sémaphore pour garantir l'accès en exclusion mutuelle.
`probOwner`	Propriétaire probable de la page.
`précédent`, `suivant`	Pointeurs sur les nœuds précédent et suivant dans la liste des nœuds qui partagent la page.
`accès`	Type d'accès autorisé sur ce nœud pour la page.



SI : Serveur d'invalidation

SRL : Serveur de recherche de pages en lecture **SDPL :** Serveur de défauts de pages en lecture
SRE : Serveur de recherche de pages en écriture **SDPE :** Serveur de défauts de pages en écriture

Figure 6.9 Architecture du gestionnaire mémoire.

6.3.6.1 Serveurs de défauts de pages

Les serveurs de défauts de pages en lecture et en écriture s'activent lorsqu'un processus génère une violation d'accès sur une page. Trois opérations peuvent produire une violation d'accès : accès WEAK sur une page qui n'est pas dans la mémoire locale du nœud, accès READ sur une page qui peut être dans la mémoire locale mais avec un droit d'accès WEAK, et finalement, un accès WRITE sur une page qui n'est pas dans la mémoire locale ou qui y est mais avec un droit d'accès READ ou WEAK.

La figure 6.10 détaille l'algorithme du serveur de défauts de pages en lecture. La procédure `Envoyer (id, mess)` utilisée dans les algorithmes envoie le message `mess` au nœud identifié par `id`. `P()` et `V()` sont les opérations classiques sur les sémaphores.

Serveur de défauts de page en lecture (SDPL) :
/* défaut de page sur la page `pag` */

```
Si pag p PTable Alors
   Prendre une entrée pour pag dans PTable
   /* Soit p cette entrée */
   /* La requête est envoyée vers le propriétaire original */
   PTable[p].probOwner ← fonction_hachage (pag)
FinSi

P (PTable[p].lock)

   Envoyer (PTable[p].probOwner, requête pag)
   Recevoir (page) /* la pagep est installée en mémoire */
   PTable[p].probOwner ← page.owner
   Si requête.accès = READ Alors
      /* Insertion dans la liste chaînée */
      PTable[p].précédent ← page.précédent
      PTable[p].suivant ← page.suivant
      PTable[p].accès ← READ
   Sinon
      PTable[p].accès ← WEAK
   FinSi

V (PTable[p].lock)
```

Figure 6.10 Serveur de défauts de page en lecture.

L'algorithme du serveur de défauts de page en écriture est présenté dans la figure 6.11. Le nœud qui reçoit un page après un défaut en écriture devient le propriétaire de la page et provoque l'invalidation de toutes les copies en accès READ. La fonction invalider envoie deux requêtes en parallèle sur la liste des nœuds qui partagent la page, et reste en attente d'un acquittement d'invalidation du dernier nœud dans la liste (premier nœud à recevoir deux requêtes d'invalidation). Toute autre requête concernant cette page sera différée en attendant l'invalidation de celle-ci sur toute la liste.

Serveur de défauts de page en écriture (SDPE) :

```
/* défaut de page sur la page pag */

Si pag ∉ PTable Alors
   Prendre une entrée pour pag dans PTable
   /* Soit p cette entrée */
   PTable[p].probOwner ← fonction_hachage (pag)
FinSi

P (PTable[p].lock)
   Envoyer (PTable[p].probOwner, requête pag)
   Recevoir (page) /* la pagep est installée en mémoire */
```

```
   /* Soit moi l'identification du nœud courant, il est le nouveau propriétaire de la page
*/
   PTable[p].probOwner ← moi
   PTable[p].accès ← WRITE
   Invalider (page.précédent, page.suivant)
   /* Après réception de l'acquittement d'invalidation */
   PTable[p].précédent ← PTable[p].suivant ← NIL

V (PTable[p].lock)
```

Figure 6.11 Serveur de défauts de page en écriture.

6.3.6.2 Serveurs de traitement des requêtes de recherche de pages

Le serveur de traitement d'une requête de recherche d'une page est exécuté par tous les nœuds qui reçoivent une requête pour la recherche d'une page en lecture ou en écriture. Pour une page donnée, ceux-ci correspondent aux nœuds dans la suite des nœuds probOwner vers le propriétaire de la page, auxquels sont ajoutés les nœuds dans la route empruntée par les messages entre ces nœuds probOwner. Rappelons-nous que seul le propriétaire d'une page peut répondre aux requêtes des pages qui demandent l'accès WRITE, alors que n'importe quel serveur avec une copie de la page compatible avec l'accès demandé peut répondre aux requêtes avec l'accès READ ou WEAK.

La figure 6.12 détaille l'algorithme du serveur de recherche de pages en lecture. Dans les serveurs de recherche de pages un contrôle additionnel est nécessaire pour éviter qu'une requête ne revienne sur le nœud qui l'a générée. Cette situation est possible à cause des modifications qui peuvent être effectuées pendant le routage. Un nœud qui a déjà traité une requête parce qu'il participait à son routage, peut faire partie de la liste des nœuds probOwner qui mène vers le propriétaire de la page (seuls nœuds à modifier la destination d'une requête). Si après une telle modification, on adresse la requête à ce nœud, il aura dans son champ probOwner l'identification du nœud qui est à l'origine de la requête et non pas celle du nœud suivant dans la liste vers le propriétaire de la page. La seule façon de récupérer la trace de la page est alors de faire suivre la requête vers le propriétaire original de la page.

Serveur de recherche de pages en lecture (SRL) :

```
/* Réception d'une requête de recherche pour la page pag */

Si pag ∉ PTable Alors

   /* Soit moi l'identification du nœud courant */
   Si requête.probOwner = moi Alors
```

```
        requête.probOwner ← fonction_hachage (pag)
        Envoyer (requête.probOwner, requête pag)
   Sinon  /* Nœud dans le chemin de routage de la requête */
      Laisser continuer la requête vers sa destination
   FinSi
Sinon

   /* Soit p l'entrée de pag dans PTable */

   P (PTable[p].lock)

   Si PTable[p].accès < requête.accès Alors
      /* NIL < WEAK < READ < WRITE * /
      Si requête.probOwner = moi Alors
         Si PTable[p].probOwner = requête.origine Alors
            /* La requête revient sur le nœud qui est à son origine */
            requête.probOwner ← fonction_hachage (pag)
         Sinon
            requête.probOwner ← PTable[p].probOwner
         FinSi
      FinSi
      Envoyer (requête.probOwner, requête pag)
      /* Idem pour les nœuds sur le chemin du routage de la requête */
      Si fonction_hachage (p) ≠ moi Alors
      /* Ce nœud n'est pas le propriétaire original de la page */
         Si requête.accès = READ Alors
            PTable[p].probOwner ← requête.origine
         FinSi
      FinSi
   Sinon
      Si requête.accès = READ Alors
         page.précédent ← moi
         page.suivant ← PTable[p].suivant
         Connecter (PTable[p].suivant)
         PTable[p].suivant ←requête.origine
         Si PTable[p].accès = WRITE Alors
            PTable[p].accès ← READ
            /* Puisqu'il existe une copie, le propriétaire perd le droit d'écriture sur la page
*/
         FinSi
      FinSi
      Envoyer (requête.origine, page pag)
   FinSi

   V (PTable[p].lock)
```

```
FinSi
```

Figure 6.12 Serveur de recherche de pages en lecture.

La procédure d'insertion d'un nœud dans la liste chaînée nécessite également qu'un serveur sur chaque nœud soit prêt à recevoir des requêtes de type connexion. A la réception d'une requête de connexion, le nœud bloque l'accès à l'entrée dans PTable de la page et modifie l'un des pointeurs de la liste. Toute autre requête concernant cette page sera différée en attendant la mise à jour de la liste. La fonction connecter envoie une requête de connexion au nœud suivant dans la liste des nœuds qui accèdent la page. Cette opération est une simple manipulation de listes, nous ne présentons pas son algorithme.

Les instructions qui doivent être exécutées pendant le routage sont intégrées aux algorithmes des serveurs de recherche de pages. Nous pouvons remarquer que le surcoût introduit par ces instructions est proportionnel à la probabilité que la page recherchée se trouve dans un nœud. En effet, les vérifications à effectuer sont réalisées de façon à ce que si la page n'est pas dans PTable, la requête est renvoyée aussitôt vers sa destination originale ; en revanche si elle a une entrée dans PTable, d'après les conditions définies pour qu'une page ait une entrée dans ce tableau, il est fort probable que le nœud dispose d'une copie de la page. Seulement dans ce cas la requête est retirée du niveau de routage (libération des liens de communication) et transmise au serveur de recherche de pages approprié.

La figure 6.13 présente le serveur de recherche de pages en écriture. Il peut arriver que le propriétaire d'une page ne dispose pas d'une copie de la page. Cela voudrait dire que la mémoire physique des nœuds du cluster n'est pas suffisante pour contenir toutes les informations manipulées par la Ptâche. Si tel est le cas, le gestionnaire continue la recherche dans l'ordre spécifié par la hiérarchie de mémoire définie dans la section 6.3.1. Des serveurs spécifiques à chaque niveau sont en mesure de recevoir ces requêtes et de les traiter, soit par l'envoi de la page recherchée si elle se trouve dans leur domaine de recherche, soit par le renvoi de la requête au niveau suivant.

Serveur de recherche de pages en écriture (SRE) :

```
/* Réception d'une requête de recherche pour la page pag */

Si pag ∉ PTable Alors

    /* Soit moi l'identification du nœud courant */
    Si requête.probOwner = moi Alors
          requête.probOwner ← fonction_hachage (pag)
          Envoyer (requête.probOwner, requête pag)
    Sinon /* Nœud dans le chemin de routage de la requête */
```

```
      Laisser continuer la requête vers sa destination
   FinSi
Sinon

   /* Soit p l'entrée de pag dans PTable */

   P (PTable[p].lock)

   Si PTable[p].probOwner ≠ moi Alors
      /* Ce nœud n'est pas le propriétaire de la page */
      Si requête.probOwner = moi Alors
         Si PTable[p].probOwner = requête.origine Alors
            /* la requête revient sur le nœud qui est à son origine */
            requête.probOwner ← fonction_hachage (pag)
         Sinon
            requête.probOwner ← PTable[p].probOwner
         FinSi
      FinSi
      Envoyer (requête.probOwner, requête pag)
      /* Idem pour les nœuds sur le chemin du routage de la requête */
      Si fonction_hachage (p) ≠ moi Alors
      /* Ce nœud n'est pas le propriétaire original de la page */
            PTable[p].probOwner ← requête.origine
      FinSi
   Sinon
      PTable[p].accès ← READ
      PTable[p].probOwner ← requête.origine
      Envoyer (requête.origine, page pag)
   FinSi

V (PTable[p].lock)

FinSi
```

Figure 6.13 Serveur de recherche de pages en écriture.

Notez aussi que le propriétaire n'invalide pas tout de suite sa copie de la page, mais il met le droit d'accès à READ. En effet, comme il est dans la liste des nœuds qui partagent la page, il recevra la requête d'invalidation générée par le nouveau propriétaire, et c'est à ce moment que l'accès sera invalidé (cf. serveur d'invalidation). La page reste alors accessible à ce nœud le temps de recevoir cette requête d'invalidation. L'invalidation de sa copie avant de recevoir la requête explicite du nouveau propriétaire de la page générerait un conflit avec le serveur d'invalidation (cf. section suivante).

6.3.6.3 Serveur d'invalidation de pages

Le serveur d'invalidation de pages, comme son nom l'indique, traite les requêtes d'invalidation en provenance du nouveau propriétaire d'une page lors d'un changement de propriétaire (voir figure 6.14). Un accès WEAK dans PTable pour une page que le serveur doit invalider signifie que le message d'invalidation a déjà traité cette requête et que celle-ci a déjà fait le tour de la liste circulaire de l'ensemble de nœuds qui partagent la page. Bien que deux nœuds puissent se retrouver dans ce cas, un protocole simple (p. ex. le nœud avec l'identification la plus petite) permet d'envoyer un seul acquittement au nouveau propriétaire de la page.

Serveur d'invalidation (SI) :

```
/* Invalidation de la page pag, soit p l'entrée de pag dans PTable */

P (PTable[p].lock)

Si PTable[p].accès = WEAK Alors
   /* La page a déjà été invalidée sur ce nœud */
   Envoyer (PTable[p].probOwner, acquittement pag)
Sinon
   Si requête.origine = PTable[p].précédent Alors
      Envoyer (PTable[p].suivant, requête pag)
   Sinon
      Envoyer (PTable[p].précédent, requête pag)
   FinSi
   PTable[p].accès ← WEAK
FinSi

V (PTable[p].lock)
```

Figure 6.14 Serveur d'invalidation.

6.3.7 Caractéristiques de l'algorithme

Notre gestion de pages utilise un algorithme de distribution dynamique extensible pour retrouver les pages dans un système à mémoire virtuelle partagée. Dans [LH89] K. Li et P. Hudak démontrent que l'algorithme de distribution dynamique est correct, c'est-à-dire, qu'il existe toujours une suite de processeurs probOwner qui mène au propriétaire de la page, et que la longueur de cette liste est au maximum de N-1 processeurs si le réseau comporte N processeurs au total.

La démonstration faite par Li et Hudak peut être extrapolée à notre algorithme ; plutôt que de refaire cette démonstration pour le cas exacte de notre algorithme, nous préférons d'analyser les

caractéristiques qui font les différences dans notre solution. Nous montrons alors que dans le pire des cas la longueur de la liste de processeurs probOwner est fortement réduite grâce à l'exploitation du routage dans la recherche d'une page.

Au début de l'algorithme, la fonction de hachage utilisée pour la distribution initiale permet à tous les processeurs de connaître le propriétaire exacte de chaque page, nous disons alors que tous les processeurs se trouvent à une distance 1 de la page. Lors d'un changement du propriétaire d'une page, tous les processeurs qui ne sont pas informés du changement s'éloignent de 1 de la page ; dans notre algorithme ceux-ci correspondent aux processeurs qui ne "voient" pas passer la requête de recherche, c'est-à-dire, des processeurs qui ne se trouvent ni sur la liste des processeurs probOwner ni sur le chemin de routage de la requête. L'objectif de cette section est montrer que la distance maximale à laquelle se trouve un processeur d'une page est bien inférieure à N pour un réseau d'une topologie quelconque de N+1 processeurs.

Soit D la distance moyenne entre deux processeurs de ce réseau, c'est-à-dire, le nombre moyen de processeurs qui doit traverser un message pour être acheminé du processeur A au processeur B. Lorsqu'un processeur envoie une requête de recherche de page, la probabilité qu'un processeur donné soit dans le chemin de cette requête est $\frac{D}{N}$. De ce fait, dans un groupe de X processeurs, $X\frac{D}{N}$ voient passer la requête alors que $X\left(\frac{N-D}{N}\right)$ ne la voient pas.

Soit S_i le nombre de processeurs qui se trouvent à une distance *i+1* du propriétaire de la page *p*, après i changements de propriétaire de p. Ceux-ci sont donc des processeurs qui après *i* changements de propriétaire de la page *p* n'ont "vu" passer aucune de ces requêtes de changement.

Nous avons alors que :

$S_0 = N$ Au début, tous les processeurs connaissent le vrai propriétaire de la page, et ils se trouvent à distance 1.

$S_1 = N - D - 1$ Après 1 changement, le processeur à l'origine de la requête, plus D processeurs ont "vu" passer la requête de recherche. Nous avons donc que N-D-1 processeurs restent à distance 2.

Afin de rester dans le pire des cas, nous considérons que la requête de changement de page nécessaire pour passer du stade S_i au stade S_{i+1}, émane d'un des processeurs qui n'ont jamais eu connaissance de ces changements. C'est cette condition qui force l'éloignement des autres processeurs de la page.

$S_2 = ((N-D-1)-1)\left(\frac{N-D}{N}\right)$ Le processeur qui demande la page, ainsi que $\frac{D}{N}$ de ceux qui restent abandonnent l'ensemble S_1. Nous pouvons réécrire S_2 :

$$S_2 = (N-D-2)\left(\frac{N-D}{N}\right)$$

Ensuite,

$S_3 = \left((N-D-2)\left(\frac{N-D}{N}\right)-1\right)\left(\frac{N-D}{N}\right)$, ce qui donne :

$$S_3 = (N-D-2)\left(\frac{N-D}{N}\right)^2 - \left(\frac{N-D}{N}\right)$$

De manière générale, pour $i > 0$:

$$\boxed{S_i = (N-D-2)\left(\frac{N-D}{N}\right)^{i-1} - \sum_{j=1}^{i-2}\left(\frac{N-D}{N}\right)^j} \quad \text{ou}$$

$$\boxed{S_i = \frac{\left(\frac{N-D}{N}\right)^{i-1}\left(D\cdot(N-D-2)+N\right)-N}{D}+1}$$

Le pire des cas pour un processeur c'est lorsqu'il se trouve à distance $i+1$ de la page, alors que tous les autres sont à une distance inférieure. Cette situation peut être modélisée comme $S_i = 1$. Dans l'algorithme de Li et Hudak on peut avoir $i = N$ pour un réseau de $N+1$ processeurs. Pour trouver la longueur maximale de la liste de probOwner nous devons trouver i tel que :

$$S_i = \frac{\left(\frac{N-D}{N}\right)^{i-1}\left(D\cdot(N-D-2)+N\right)-N}{D}+1 = 1 \quad (1)$$

De (1) nous pouvons obtenir la valeur de i :

$$i = \frac{\log\left(\frac{N}{D\cdot(N-D-2)+N}\right)}{\log\left(\frac{N-D}{N}\right)}+1$$

Nous pouvons maintenant calculer la valeur de i pour des réseaux de processeurs de différente taille. La valeur de D est une fonction de la topologie du réseau et du nombre de processeurs. La distance moyenne des réseaux réguliers a été calculée pour ces réseaux. Nous pouvons aussi

vérifier que la plupart des processeurs sont près d'une page par rapport à la longueur maximale de la liste de probOwner. Le tableau 6.1 présente quelques exemples de la valeur de *i* pour la longueur maximale ($S_i = 1$) et pour $S_i = \frac{N}{2}$, c'est-à-dire, *i* tel que la moitié des processeurs du réseau se trouvent à une distance inférieure à *i*. Nous utilisons plusieurs valeurs de D pour l'exemple.

N	D	i, $S_i = 1$	i, $S_i = \frac{N}{2}$
100	2	54	19
100	5	34	9
1,000	3	461	155
1,000	10	238	59
10,000	4	4,022	1276
10,000	20	1,520	322
100,000	5	35,834	10,778
100,000	40	9,282	1671

Tableau 6.1 Analyse du pire des cas dans un réseau de N processeurs et distance moyenne D.

6.4 Conclusion

L'intérêt d'un espace d'adressage commun dans une architecture à mémoire distribuée est amplement reconnu. D'une part, il permet l'exploitation du paradigme de programmation basé sur les variables partagées, ce qui met à la portée d'un public plus large l'utilisation de ces architectures. D'autre part, et c'est le point qui nous intéresse le plus dans notre travail, il permet l'utilisation des mémoires locales des différents nœuds de la machine comme un cache de la mémoire secondaire grâce à la projection des fichiers dans l'espace d'adressage virtuel.

La résolution d'un défaut de page est particulièrement ardue dans une architecture extensible à mémoire distribuée. Nous avons donc tout d'abord défini une hiérarchie de mémoires afin de

limiter l'espace de recherche d'une page. Ensuite, nous avons proposé un protocole de recherche des pages dans cette hiérarchie qui, afin d'améliorer ses performances, exploite directement la couche de routage du micro-noyau ParX.

La participation du routage pendant la recherche d'une page vise à raccourcir le chemin d'accès à une page et à réduire le nombre de messages générés par une recherche. Ces deux propriétés sont fondamentales dans les architectures massivement parallèles car leurs performances sont fortement dépendantes des communications inter-processeurs. Néanmoins, l'utilisation du routage pour la recherche d'une page est une fonctionnalité qui ne peut être exploitée que lorsque le routage est fait dans le nœuds de calcul et que le système d'exploitation peut contrôler ce routage si celui-ci s'appuie sur des mécanismes matériels. La gestion de la mémoire virtuelle est alors un exemple des fonctionnalités système qui bénéficierait du rajout de ce contrôle dans les circuits de routage automatique.

Une fois limitée la taille du tableau PTable nécessaire aux protocoles de recherche et de cohérence des pages, nous obtenons un gestionnaire de pages pour un système à mémoire virtuelle partagée qui peut être appliqué aux systèmes massivement parallèles. Dans l'objectif de servir de base à un système d'E/S, le modèle de cohérence retenue combine la cohérence forte et la cohérence faible. Ainsi, le gestionnaire mémoire est capable de garantir la cohérence forte sur un fichier pour un ensemble de processus, et en même temps, de laisser à d'autres l'accès sur des blocs de ce fichier qui ne sont peut être plus à jour.

Chapitre 7
Conception d'un système de fichiers massivement parallèle

Au chapitre 3 nous avons présenté quelques exemples de SF parallèles et passé en revue les principaux projets de recherche sur les SF pour les architectures massivement parallèles. De cette présentation, nous retiendrons que pour obtenir les meilleures performances, la tendance actuelle dans la conception des SF est de cibler un ensemble précis d'applications, et surtout des applications qui dans chaque opération d'E/S le transfert de données est d'une taille importante. Nous qualifierons ces SF adaptés à un type particulier d'applications, de *SF spécialisés.*

Comme nous l'avons précisé dès l'introduction, nous nous intéressons aux systèmes universels, c'est-à-dire destinés à tout type d'applications, quels que soient leurs besoins en E/S. Ce besoin de généralité est d'autant plus important que nous acceptons qu'un même ensemble de données puisse être accédé par plusieurs applications de façons radicalement différentes. Le SF doit alors être assez général pour traiter tout type de requête, mais il doit également prévoir des optimisations en fonction des requêtes afin d'offrir les meilleures performances aux applications.

Ce chapitre définit une architecture logicielle de SF parallèle, qui a comme objectif d'offrir aux applications des accès performants aux données stockées en mémoire secondaire. L'architecture proposée ne fait aucune supposition sur le type attendu des requêtes d'E/S et est conçue avec un souci particulier d'extensibilité afin de pouvoir l'appliquer aux architectures massivement parallèles.

Nous commençons par introduire les principes de base qui doivent être respectés pour concilier universalité et hautes performances dans la conception d'un SF. Puis, nous faisons une présentation détaillée du SF massivement parallèle conçu d'après ces principes.

7.1 Directives pour la conception d'un système de fichiers massivement parallèle

Efficacité et universalité ont été traditionnellement dissociées dans les systèmes informatiques, comme si atteindre de hautes performances et être capable de traiter un ensemble d'applications le plus vaste possible étaient contradictoires. La tendance actuelle dans la conception des SF parallèles est de perpétuer cette pratique ; et c'est ce qui explique pourquoi plusieurs des SF proposées aujourd'hui sont des SF spécialisés.

Dans les architectures massivement parallèles, une contrainte vient s'ajouter à la difficulté de réconcilier ces deux concepts : l'extensibilité. La possibilité d'ajouter ou de diminuer les ressources physiques d'une machine parallèle, tout en conservant un équilibre dans la puissance de la machine, pose des problèmes nouveaux pour la conception et l'implémentation des sous-système d'E/S.

Nous pensons qu'il est possible de réaliser un SF universel et performant dans un environnement extensible si l'on respecte quelques directives lors de sa conception. Pour ce faire, il est nécessaire d'y faire participer le système d'exploitation et les utilisateurs, chacun à un niveau approprié. Seulement une bonne coopération entre le SF, le système d'exploitation et les utilisateurs permet l'existence d'une couche logicielle efficace pour les E/S, et garantit un équilibre entre les services généraux et les fonctionnalités spécifiques.

Dans cette section nous identifions les principes de base qui doivent être respectés pour concevoir un SF massivement parallèle. Evidemment, l'extensibilité est notre souci principal mais deux autres aspects nécessitent une attention particulière : l'extension des interfaces standards pour ajouter de nouvelles fonctionnalités aux sous-systèmes d'E/S, et l'exploitation du parallélisme à chaque étape d'une opération d'E/S.

7.1.1 L'extensibilité

Comme nous l'avons précisé dans le chapitre 2 le concept d'extensibilité doit être étendu aux E/S. Pour qu'une architecture soit vraiment extensible, il ne suffit pas de moduler uniquement la puissance de calcul, il faut aussi que la puissance en E/S soit adaptable aux besoins des applications.

Cependant, l'extensibilité du matériel ne signifie rien si les logiciels développés pour l'exploitation de ce matériel ne sont pas eux aussi extensibles. Deux volets composent

l'extensibilité d'un logiciel. En premier lieu, la capacité de celui-ci à s'exécuter sur une architecture indépendamment des ressources physiques qui lui sont allouées (nombre de processeurs, mémoire vive, etc.). En deuxième lieu, les performances obtenues par le logiciel doivent être en rapport avec la puissance potentielle de l'architecture. Il ne sert à rien d'ajouter des éléments à une architecture si les applications exhibent toujours les mêmes performances. Un SF extensible doit savoir tirer le meilleur profit de toutes les unités de calcul et de stockage, et très particulièrement, de toutes les voies d'E/S mises à sa disposition.

Nous distinguons quatre priorités fondamentales pour garantir l'extensibilité d'un SF parallèle : éviter l'utilisation de données centralisées, rendre les structures internes du SF indépendantes de l'architecture matérielle sous-jacente, baser le SF sur un micro-noyau extensible, et rapprocher le plus possible les différents serveurs du SF de leurs clients pour renforcer la localité des requêtes. Nous décrivons en détail chacune de ces caractéristiques.

7.1.1.1 Données non centralisées

La conception d'un logiciel extensible doit éviter l'existence de toute donnée centralisée. Aucun service d'un SF extensible ne doit être assuré par un seul et unique serveur dans le système. En effet un service centralisé dans une architecture où le nombre de clients peut être très important devient rapidement un goulot d'étranglement du système.

Tout serveur doit alors être répliqué selon les critères suivants. D'une part, le nombre de serveurs pour un service doit être en rapport avec la taille de l'architecture. Dans notre architecture générale définie dans la section 2.3.3, ceci signifie que ce nombre est une fonction des quatre paramètres qui caractérisent l'architecture : *p*, le nombre de processeurs de calcul ; *i*, le nombre de processeurs d'E/S ; *c*, le nombre de contrôleurs et *d*, le nombre de disques par contrôleur. D'autre part, il faut que le service proposé par chacune des instances d'un serveur puisse être assuré indépendamment des autres : si les différentes instances d'un serveur partagent des données entre elles, le surcoût pour maintenir à jour ces données devient critique.

Les fichiers entrelacés ont été proposés pour distribuer les données sur plusieurs unités de stockage, de manière à ce qu'un fichier soit accessible en parallèle : des requêtes à un même fichier sont adressées à des serveurs différents dans les sous-système d'E/S. En général, les SF parallèles existants utilisent une distribution pour les données d'un fichier basée sur la politique du tourniquet, mais des approches plus flexibles ont été aussi proposées (cf. le SF holographique de la section 4.2.4.2).

Nous pouvons appliquer aux données internes du SF la distribution opérée sur les fichiers. La structure arborescente des répertoires peut également être distribuée sur plusieurs serveurs de telle façon à ce que les requêtes d'ouverture de fichiers s'adressent à différents serveurs (en fonction du nom du fichier par exemple). Par analogie nous appelons de tels répertoires : *répertoires entrelacés*.

7.1.1.2 Structures indépendantes de l'architecture

La définition d'une architecture matérielle générale est importante pour notre travail parce qu'elle nous permet de rester indépendant vis-à-vis des possibles caractéristiques spécifiques de certaines machines. Cette généralité est obtenue par abstraction de la topologie des réseaux d'interconnexion et des paramètres de l'architecture (p, i c, d). La seule hypothèse sur laquelle la conception peut s'appuyer est la relation entre les valeurs de ces paramètres, car comme nous l'avons énoncé au deuxième chapitre, pour obtenir une extensibilité globale, les valeurs de ces paramètres sont dépendantes les unes des autres.

Les structures internes nécessaires à l'exécution du SF doivent rester indépendantes de l'architecture sous-jacente, et la taille de la mémoire nécessaire pour représenter ces structures doit être limitée par la mémoire disponible. Bien qu'il soit naturel que la taille des structures internes nécessaires à la gestion d'un service augmente lorsque la taille de l'architecture augmente elle aussi, il faut que la courbe de croissance de la première soit bornée par celle de la croissance de l'architecture.

Pour le cas spécifique des SF ceci se reflète dans la taille des structures en mémoire vive mais également dans la taille des données nécessaires sur disque pour la gestion des fichiers. Lors de l'extension d'une machine parallèle, la taille des données maintenues en mémoire vive par le SF ne peut pas augmenter plus que la mémoire vive rajoutée et la taille des méta-données stockées sur disque (répertoires et autres informations associées aux fichiers) ne peut pas augmenter plus que le nouvel espace disque disponible.

7.1.1.3 Micro-noyau extensible

La caractéristique d'extensibilité d'un logiciel doit être garantie dès la base. Aujourd'hui le micro-noyau est la base des SF dans les systèmes d'exploitation parallèles ; outre la gestion des processus et de la mémoire, le rôle principal du micro-noyau est d'offrir les mécanismes de base pour la communication inter-processus.

La communication inter-processus est fondamentale pour le SF. D'une part, dans un environnement distribué les appels système pour demander les services du SF sont transformés en messages qui sont acheminés via les mécanismes de communication inter-processus du micro-noyau vers le(s) serveur(s) de fichiers. D'autre part, les données qui concernent ces demandes sont elles aussi transférées à travers le réseau d'interconnexion grâce à ces mêmes mécanismes. Par ailleurs, la réalisation d'un service du SF peut nécessiter des protocoles spécifiques qui n'ont pas été prévus dans le micro-noyau. Bâtir ces protocoles sur les protocoles de base du micro-noyau induit un surcoût inutile. Il est alors souhaitable que le micro-noyau soit capable d'intégrer, d'une façon constructive ou générique, de nouveaux protocoles au même niveau que les protocoles de communication de base.

7.1.1.4 Maintien de la localité

L'extension d'une architecture peut éloigner certaines ressources de leurs clients potentiels, ce qui peut entraîner des pertes des performances car l'accès à des ressources distantes est plus coûteux que l'accès à des ressources locales. Une façon d'améliorer l'extensibilité est alors de restreindre le domaine de communication de chaque processus pour limiter les demandes à des serveurs distants. Ceci améliore les performances globales des applications parallèles car le risque de contention entre messages indépendants est diminué, ainsi que le temps de réponse pour les requêtes qui peuvent être traitées localement.

Placer les serveurs aussi près que possible de leurs clients potentiels est une façon de préserver la localité. Les structures de données peuvent aussi être placées judicieusement pour limiter les consultations à l'extérieur d'un domaine de communication. Pour le SF, la gestion de caches des fichiers accédés par une application doit alors se faire à l'intérieur du domaine de communication de l'application, et les méta-données d'un fichier doivent être maintenues près du disque qui contient les données du fichier.

La localité dans l'accès aux données d'un fichier est une caractéristique très fréquente des applications séquentielles. Puisque les applications parallèles sont composées de processus séquentiels, le SF peut en profiter pour développer des politiques qui exploitent cette localité. De plus, à l'intérieur du domaine de communication d'une application, ces politiques peuvent être optimisées en fonction du type d'accès fait par l'application.

7.1.2 Extension des interfaces standard

Dans [Kotz92], D. Kotz décrit les inconvénients d'utiliser une interface de système de fichiers de type Unix pour les machines parallèles. Dans son étude, l'auteur considère seulement les accès aux fichiers en lecture exclusive séquentielle et en écriture exclusive séquentielle, mais ces exemples sont suffisants pour nous donner une idée des problèmes que rencontrent les applications, faute d'une interface plus évoluée, pour exprimer leurs besoins et/ou pour exploiter les capacités des nouveaux sous-système d'E/S.

7.1.2.1 L'interface conventionnelle et ses problèmes

L'interface du système de fichiers d'Unix est un bon exemple d'interface conventionnelle. Cette interface est utilisée actuellement par la plupart des systèmes séquentiels, distribués, et même parallèles.

Pour Unix un fichier est simplement une suite d'octets. Les opérations offertes pour le traitement d'un fichier sont : ouverture, création, fermeture, lecture, écriture et une opération de positionnement. Ce modèle simple, qui n'a pas été prévu pour des machines parallèles, présente plusieurs problèmes pour être appliqué à ces machines.

- *Partage de fichiers ouverts* : tout processus qui veut accéder à un fichier est contraint d'exécuter une opération d'ouverture (`open`). Dans Unix, le partage de fichiers par héritage des fichiers ouverts (`fork()`) est limité et n'est applicable qu'aux processus qui possèdent un ancêtre commun. Pour effectuer l'ouverture d'un fichier tous les processus sont obligés de connaître le nom du fichier (et les droits correspondants de lecture/écriture), mais plus grave encore, une opération d'ouverture par processus d'une application parallèle peut générer une surcharge considérable sur le système de fichiers (le SF doit répondre séparément à chaque demande).

- *Accès coopératif aux fichiers* : lors de l'ouverture d'un fichier par plusieurs processus, chaque processus obtient un pointeur local sur le fichier. L'implantation de mécanismes de coopération entre processus qui accèdent aux mêmes fichiers devient alors très difficile. Bien qu'il existe des mécanismes qui permettent le partage des pointeurs de fichiers, ils sont très primitifs et le contrôle de la concurrence dans les machines sans mémoire commune est très complexe pour exécuter les opérations de lecture et d'écriture de façon atomique.

- *Fichiers entrelacés* : les SF classiques n'offrent pas une vue transparente à l'utilisateur des fichiers entrelacés. Ceci est un handicap pour les applications qui devront se soucier de la localisation physique de chaque bloc du fichier.

- *Segmentation de fichiers* : une autre opération difficile à exécuter avec l'interface classique est le traitement d'un fichier par segments. Dans ce cas, un fichier est divisé en plusieurs parties appelées segments, et chaque segment est traité par un processus différent d'une application parallèle. Le SF doit alors gérer, de façon globale, la distribution des segments

du fichier à chaque processus, et pour les opérations d'écriture, récolter les morceaux pour les rassembler dans une seule unité physique.

- *Utilisation de tampons* : si les tampons de niveau utilisateur sont alloués pour le couple processus-fichier, des problèmes de cohérence peuvent se manifester car les données peuvent évoluer différemment sur chaque processeur alloué à l'application. Une gestion des caches des différents processus d'une application parallèle est alors nécessaire.

Afin de faciliter aux programmeurs l'exploitation des avantages des E/S parallèles, il est nécessaire de leur proposer des interfaces de plus haut niveau pour l'accès aux fichiers. Cependant, il est aussi nécessaire que ces interfaces soient des extensions de l'interface traditionnelle car celle-ci est indispensable à un vaste ensemble de modèles de programmation : le modèle d'E/S d'Unix représente le minimum de services requis par une grande part des applications. Par ailleurs, la compatibilité avec les interfaces existantes permet aux applications qui les utilisent de profiter des avantages des E/S parallèles (p. ex. pour faciliter le portage des applications). En effet, l'implémentation des primitives de lecture et d'écriture peut profiter de la distribution physique des données, tout en cachant cette distribution aux applications.

7.1.2.2 Caractéristiques utiles aux E/S parallèles

Une première réflexion sur les types de fonctionnalités dont les applications parallèles ont besoin a été conduite à Dartmouth College [Ko92, Ko94], et ses résultats ont été le point de départ du projet CHARISMA [KN94, PEKN95]. CHARISMA étudie le comportement des applications du point de vue des E/S pour déterminer les primitives qui doivent être supportées par un SF parallèle. Néanmoins, cette étude se limite aux applications de calcul scientifique et aujourd'hui un nouveau projet de définition d'interfaces, MPI-IO [MPIO95], a démarré au sein des concepteurs de la bibliothèque standard MPI [MPI94]. La grande nouveauté du projet MPI-IO est d'être un forum ouvert au grand public à travers Internet, ce qui a permis d'exprimer des intérêts très divers en matière d'E/S.

Dans cette section nous résumons les concepts qui ont été identifiés comme des caractéristiques souhaitables pour un SF parallèle.

- *Une structure de répertoires pour l'espace de noms* : une vision de type Unix semble appropriée car elle cache les aspects physiques de l'implémentation. Si un fichier ou le contenu d'un répertoire est distribué sur plusieurs disques, ceci doit être entièrement transparent à l'utilisateur.

- *Accès indépendant aux différents blocs d'un fichier* : chaque processus doit avoir la possibilité d'accéder à un fichier indépendamment des accès faits par les autres processus. Il est même souhaitable que chaque processus ait une vue propre d'un fichier avec un pointeur sur des adresses relatives à sa vue.

- *Multi-ouverture (multiopen)* : le SF doit offrir une opération multiopen qui ouvre un fichier pour tous ou un sous-ensemble des processus d'une application parallèle. Cette opération n'ouvre le fichier qu'une seule fois et elle communique, ensuite à chaque processus participant à l'ouverture, un descripteur de fichier pour l'accès aux données.

- *Pointeur de fichier* : un processus doit pouvoir indiquer que le pointeur du fichier retourné par l'opération de multiopen est local ou global. Un pointeur global offre la synchronisation nécessaire pour les politiques d'accès globaux : la lecture du fichier et la mise à jour du pointeur forment une seule opération atomique. L'opération de positionnement pourra s'appliquer aux deux types de pointeurs. Afin de conserver la sémantique des opérations classiques, deux nouvelles instructions sont introduites : `readp` et `writep`. Ces instructions renvoient le nombre d'octets lus, mais aussi la valeur du pointeur global avant l'exécution de l'instruction.

- *Projection des fichiers sur les processus* : des fonctions de mise en correspondance entre les pointeurs des fichiers et les positions à l'intérieur de ces fichiers s'avèrent très utiles Une telle fonction avec ses paramètres, permettra aux processus d'accéder à des portions non contiguës d'un fichier avec un seul pointeur local par fichier (cf. SF de nCUBE section 4.2.4.3).

- *Enregistrements logiques* : l'accès aux fichiers doit pouvoir se faire par enregistrements logiques de taille variable et l'atomicité des opérations doit être définie au niveau de ces enregistrements. Ceci permet un meilleur contrôle du SF sur les fichiers (par exemple éviter de diviser un enregistrement sur différents disques).

- *Multi-fichiers (multifiles)* : pour le traitement de fichiers par segments, la notion de *multi-fichier* est proposée. Un multi-fichier est un seul fichier structuré en plusieurs sous-fichiers, où chaque sous-fichier est traité par un processus différent. Une opération de correspondance entre processus et sous-fichiers doit être aussi disponible.

- *Transtypage* : la distinction entre multi-fichiers et fichiers classiques d'une part et entre fichiers à enregistrements variables et fichiers à enregistrements d'un octet d'autre part, conduit à quatre types de fichiers différents. Une application doit avoir la possibilité d'ouvrir un fichier dans un mode autre que le mode natif du fichier : on parle alors de transtypage.

7.1.2.3 Contrôle de bas niveau

Les interfaces de haut niveau sont bien adaptées pour les programmeurs qui ne veulent pas se soucier de l'implémentation interne du SF, mais qui veulent toutefois obtenir de bonnes performances pour leurs applications. Cependant, il existe des utilisateurs expérimentés qui veulent tirer le meilleur profit des architectures ; ils ont en général une bonne connaissance de l'architecture et de son système d'exploitation et ils sont capables d'exploiter au mieux leurs caractéristiques. Pour que ces utilisateurs puissent faire de même avec le SF, il faut que celui-ci leur laisse le choix dans un certain nombre de décisions. De ce fait, le SF doit être suffisamment flexible pour qu'un programme puisse contrôler les divers paramètres qui pourraient améliorer les performances. Dans [CK93], les auteurs identifient quelques uns de ces paramètres :

- *Contrôle sur l'entrelacement des fichiers* : le programmeur doit avoir la possibilité de définir la taille des blocs à entrelacer et les unités disque à utiliser ;

- *Des informations sur la configuration* : les applications peuvent avoir besoin de connaître la configuration physique exacte de l'architecture utilisée (nombre de processeurs, mémoire disponible, nombre de disques, taille d'un bloc disque, méthode d'entrelacement) ;

- *Contrôle sur la parité* : le contrôle de la parité dans un transfert du SF n'est pas toujours nécessaire ; il doit être possible pour une application de le désactiver ;

- *Contrôle sur la gestion des caches et sur l'anticipation des demandes de blocs* : certaines applications font elles mêmes la gestion des caches et connaissent la meilleure politique pour accéder aux blocs de façon à rester optimales. Ces opérations de bas niveau doivent elles aussi pouvoir être désactivées à la demande.

Comme extension d'une telle interface, une bibliothèque de services SF doit aussi être disponible. A titre d'exemple, on peut identifier plusieurs opérations fréquemment sollicitées : le tri, la copie de fichiers, la transformée de Fourier, la transposition de matrices, etc. Ces services système s'exécuteraient sur les PES et exploiteraient alors directement la structure interne du SF pour offrir les meilleures performances.

7.1.3 Exploitation du parallélisme

L'architecture matérielle des sous-systèmes d'E/S parallèles offre aux applications plusieurs voies d'accès aux données stockées en mémoire secondaire. C'est à l'architecture logicielle que revient la tâche de coordonner ces accès de façon à exploiter le plus possible toutes les voies disponibles. C'est le parallélisme des opérations sur ces voies qui permet d'écourter le temps de service des requêtes d'E/S.

Le parallélisme exprimé par une application parallèle et la possibilité d'exécuter plusieurs applications simultanément sur la même machine sont une source importante de concurrence pour l'accès aux données stockées en mémoire secondaire. Pour obtenir des accès performants, ce parallélisme doit être préservé, et si possible augmenté, à chaque étape d'une opération d'E/S.

Le parallélisme d'un SF est également obtenu à partir des fichiers et répertoires entrelacés, et grâce aux multiples voies d'accès à ces données offertes par le sous-système d'E/S. Lorsque plusieurs processus d'une application (ou des applications différentes s'exécutant simultanément) demandent des services indépendants, ils doivent être servis en parallèle. Naturellement, ceci est possible seulement si le nombre de centres de services et la concurrence disponible dans les structures de données du SF augmentent avec la taille de l'architecture.

Dans l'architecture générale proposée dans le chapitre 2, le parallélisme est présent à quatre niveaux. Premièrement, les unités de calcul peuvent générer leurs requêtes d'E/S en parallèle ; chaque UC doit pouvoir gérer ses requêtes sans que l'application ait à fournir un serveur propre pour traiter les requêtes d'E/S. Ensuite, si le SF est composé d'un nombre de serveurs au moins égal au nombre de processeurs d'E/S, la bande passante entre les contrôleurs et les unités de calcul est maximisée. Puis, les contrôleurs peuvent être accédés en parallèle à partir de n'importe lequel des serveurs sur les PES, car les PES peuvent communiquer directement avec un contrôleur quelconque sans avoir à utiliser un autre PES comme intermédiaire. Enfin, avec la technologie des disques à têtes fixes ou celle des vecteurs de disques, l'accès aux blocs physiques peut aussi être fait en parallèle.

L'organisation de la couche logicielle des E/S doit être capable de tirer profit de chacun des niveaux de parallélisme disponible dans l'architecture, et c'est au système de fichiers d'implanter les politiques nécessaires pour assurer l'équilibre entre les différentes requêtes d'E/S à travers le système.

La gestion des caches et du préchargement de données des fichiers est un domaine qui doit aussi bénéficier du parallélisme de l'architecture. Si ce problème a été bien étudié pour les architectures monoprocesseurs [OCHK85, BHKS91] ceci n'est pas le cas pour les architectures parallèles. Dans sa thèse [Ko91] et dans un article postérieur avec C.S. Ellis [KE93], D. Kotz présente une étude des différentes organisations de caches et de politiques de préchargement pour une architecture MIMD à mémoire commune, mais là encore, l'absence de connaissance sur le type d'accès aux fichiers présents dans les applications parallèles ne permet pas de tirer de conclusions définitives.

7.2 Un Système de Fichiers pour PAROS

La base d'un SF massivement parallèle doit être un micro-noyau massivement parallèle, et ParX en est un. L'architecture logicielle proposée ici est influencée par l'approche de ParX car elle exploite les différentes caractéristiques offertes par ce micro-noyau. Elle reste néanmoins générale dans le sens où l'on peut l'adapter à d'autres systèmes avec des approches différentes. Nous sommes convaincus que les concepts indispensables à sa réalisation sont ceux qui doivent être présents dans tout micro-noyau parallèle qui se veut extensible.

Une caractéristique très intéressante de ParX est qu'il laisse aux utilisateurs le choix du niveau d'abstraction (machine virtuelle) le mieux adapté à leurs besoins. Nous avons choisi d'utiliser la machine virtuelle à mémoire partagée car elle implémente déjà des fonctionnalités nécessaires à

l'accès aux données distantes, et parce qu'elle permet de bénéficier des avantages des E/S par projection des fichiers en mémoire (*memory mapping*). En effet, cette machine virtuelle nous permet d'utiliser les mémoires locales comme caches du SF, ce qui induit un surcoût inférieur à celui de la méthode traditionnelle des tampons système, et qui décharge le système de fichiers d'une partie de la gestion de cohérence des données (cf. chapitre 6).

Dans cette section nous définissons l'architecture du SF massivement parallèle pour le système d'exploitation PAROS. Avant de définir les différents composants de l'architecture du SF nous détaillons le système de désignation utilisé par notre approche.

7.2.1 Désignation

Le service de désignation assure la correspondance entre les noms symboliques des fichiers, ou noms externes, et leurs noms internes ou identificateurs uniques (UID) à partir desquels il est possible d'accéder aux données. Cette tâche doit prendre en compte la distribution des répertoires et des fichiers sur les différentes unités de disques. La distribution utilisée des blocs d'un fichier est celle d'une organisation de niveau 5 des systèmes RAID, et l'arborescence des répertoires commence par le répertoire racine (*root*) "barre oblique" ("/"). Dans cette section nous détaillons comment le SF de PAROS effectue cette désignation.

7.2.1.1 L'identificateur unique

Dans un SF parallèle, le SF est composé de plusieurs processus distribués qui s'exécutent sur les différents nœuds d'E/S de la machine, ce qui pose deux problèmes cruciaux au système de désignation de chacun de ces processus dans un environnement extensible : d'une part, comment attribuer un nouvel identificateur de manière autonome et être sûr qu'il est unique pour tout le SF ? et d'autre part, une fois l'identificateur attribué, comment identifier le ou les processus du SF qui connaissent cet identificateur à partir du nom symbolique du fichier ?

Afin de rester compatibles avec la plupart des SF proposés pour les environnements distribués et parallèles nous avons choisi d'utiliser une désignation intégrée au stockage (cf. section 4.1). Pour retrouver l'identificateur unique d'un fichier, il suffit de suivre un à un les répertoires qui composent le nom symbolique jusqu'à retrouver le répertoire où se trouve le fichier. Nous détaillons dans la section suivante, lors de la présentation des serveurs qui constituent notre SF, les démarches suivies pour diminuer les risques de faire de certains répertoires des goulots d'étranglement du système et pour accélérer le processus de traduction de noms symboliques en UIDs.

Le SF de PAROS utilise une structure hiérarchique de répertoires basée sur la notion de répertoire entrelacé. Un répertoire entrelacé est un fichier qui contient la correspondance entre le nom symbolique et l'identificateur unique de chaque fichier du répertoire. La différence avec les répertoires classiques se trouve dans la localisation des fichiers à l'intérieur du répertoire : les fichiers à l'intérieur d'un répertoire entrelacé ne sont pas nécessairement dans la même unité de stockage que le répertoire. De la même façon que les blocs d'un fichier sont distribués sur plusieurs disques selon une fonction de hachage, lors de la création d'un fichier entrelacé[1], le disque sur lequel le premier bloc du fichier doit être stocké est calculé d'après une fonction de hachage appliquée sur le nom du fichier dans le répertoire (dernière composante du nom symbolique).

Les systèmes classiques ont des SF autonomes et utilisent le numéro de l'entrée dans une table globale de fichiers comme identificateur unique des fichiers (numéro d'i-nœud dans les cas des SF du type Unix). Ceci est possible car dans ces systèmes tout fichier possède une structure unique (l'i-nœud) qui contient les informations nécessaires à un processus pour accéder à un fichier, tels que le propriétaire du fichier, les droits d'accès, la taille du fichier, et la correspondance entres les blocs logiques d'un fichier et les blocs physiques du disque. Les i-nœuds se présentent sous une forme statique sur le disque, et le SF maintient un cache d'i-nœuds en mémoire pour assurer leur gestion [Ba86].

En raison de la répartition des fichiers entrelacés sur plusieurs disques, la gestion des i-nœuds nécessite deux modifications : d'une part l'i-nœud doit prendre en compte cette distribution pour la localisation des blocs des fichiers (cf. section suivante), et d'autre part il faut trouver le disque qui convient le mieux pour abriter l'i-nœud d'un fichier donné. Puisque les informations contenues dans l'i-nœud d'un fichier sont nécessaires lors de l'ouverture du fichier, placer l'i-nœud dans le disque où se trouve le premier bloc du fichier est tout à fait naturel.

La sélection des identificateurs uniques dans un système parallèle à mémoire distribuée est une conséquence de cette gestion des i-nœuds. Nous utilisons un identificateur composé de deux éléments : l'identification du disque déterminé par la fonction du hachage sur le nom du fichier, et un identificateur local à ce disque qui est le numéro de l'i-nœud associé au fichier. L'identificateur local peut être ainsi déterminé de façon autonome par la composante du SF qui s'exécute sur le processeur d'E/S chargé de la gestion du disque identifié par la fonction de hachage.

L'UID ainsi formé répond de façon satisfaisante aux problèmes posés au mécanisme de désignation. A la création d'un fichier, une fonction de hachage détermine le disque qui doit

[1] Un fichier stocké entièrement sur un seul disque est un fichier entrelacé où l'unité d'entrelacement est le fichier lui-même.

stocker l'i-nœud et le premier bloc de données du fichier, et ensuite le processus du SF qui s'exécute sur le PES chargé de la gestion de ce disque attribue au fichier son identificateur unique sans avoir à consulter d'autres serveurs. Cet identificateur est enregistré dans le répertoire où se trouve le fichier, de façon à ce que l'on puisse le retrouver ultérieurement lors d'une opération d'ouverture du fichier.

La figure 7.1 donne un exemple de la disposition d'un répertoire entrelacé. Un répertoire est un fichier dont les données sont une suite d'éléments qui comprennent chacun un identificateur unique (couple identificateur du disque, identificateur local au disque) et le nom d'un fichier contenu dans ce répertoire. Hormis l'identification du disque, la structure du répertoire est la même que dans les SF traditionnels. Chaque répertoire contient les noms des fichiers point et point-point (“.” et “..”) dont les numéros d'i-nœud sont respectivement celui du répertoire et celui du répertoire parent. Les éléments d'un répertoire peuvent être vides, ce qui est indiqué par le couple (0,0) comme c'est le cas pour le fichier “crash” de l'exemple (fichier effacé).

Nom fichier	Disque	i-nœud
.	3	78
..	7	123
a.out	2	12
bin	4	87
crash	0	0
par.c	5	96
queue.c	2	54
tmp	6	77

Figure 7.1 Structure d'un répertoire entrelacé.

7.2.1.2 Localisation des blocs d'un fichier

Pour localiser tous les blocs qui appartiennent à un fichier nous devons tout d'abord ajouter à l'i-nœud du fichier des informations sur la distribution de ces blocs ; ensuite, il faut conserver des informations additionnelles pour faire la correspondance entre les blocs logiques du fichier et les blocs disque pour chacun des disques utilisés pour la distribution du fichier. Puisque cette correspondance entre blocs logiques et physiques est spécifique à chaque disque, elle ne fait alors plus partie de l'i-nœud du fichier, mais doit être distribuée sur chaque disque qui contient au moins un bloc de données du fichier.

L'i-nœud d'un fichier contient alors toutes les informations nécessaires pour l'ouverture d'un fichier (le propriétaire, les droits d'accès, la taille, etc.) et très particulièrement les paramètres de la fonction de hachage utilisée pour la distribution de ses blocs. En effet, parmi les contrôles de bas niveau offerts à l'utilisateur par le SF, se trouve la définition de la taille des blocs à entrelacer et les unités disque à utiliser pour répartir ces blocs. La fonction de hachage utilisée par le SF est toujours la même pour respecter une distribution de niveau 5 des systèmes RAIDs, mais ses paramètres peuvent être modifiés, soit parce que la configuration du système a changé (par exemple par modification du nombre de disques ou des PES), soit parce que l'utilisateur veut définir lui-même la façon dont la distribution du fichier doit être faite. Ces paramètres font alors partie des informations intégrées à un fichier lors de sa création.

Pour retrouver le disque qui contient un bloc spécifique d'un fichier, il suffit de le calculer grâce à la fonction de hachage à partir du numéro de bloc et avec les paramètres conservés dans l'i-nœud du fichier. Une requête de lecture ou d'écriture d'un bloc peut être alors envoyée au processus du SF chargé de la gestion du disque déterminé par cette fonction de hachage. La localisation de ce processus parmi les différents PES n'est pas un problème car la configuration du sous-système d'E/S ne change pas pendant l'exécution d'une Ptâche.

La fonction de hachage permet non seulement de préciser le disque où se trouve un bloc spécifique d'un fichier, mais également elle détermine la position qu'occupe ce bloc parmi tous les blocs du même fichier stockés sur ce disque. Par exemple pour la fonction de hachage *modulo* avec N disques ($f(bloc) = bloc \bmod N$), pour calculer la position d'un bloc dans un disque, il suffit d'ajouter 1 au résultat de diviser le numéro du bloc par le nombre de disques utilisés pour la distribution du fichier. Ainsi, le bloc 25 d'un fichier distribué sur 8 disques est le quatrième bloc du fichier dans le premier disque utilisé pour la distribution (cf. figure 7.2).

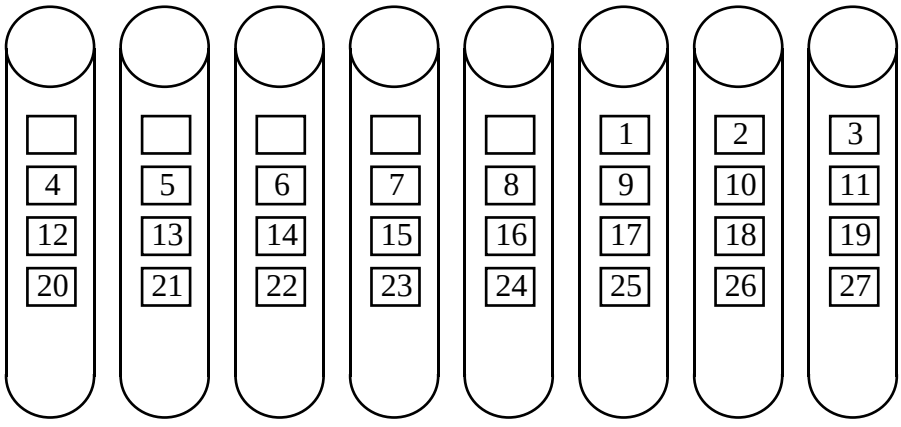


Figure 7.2 Distribution des blocs d'un fichier entrelacé.

Il ne nous reste qu'à retrouver sur le disque le bloc physique qui contient les données d'un bloc logique du fichier. Pour ce faire, sur chaque disque qui contient des données d'un fichier, le SF

conserve une cartographie du disque similaire à celle que l'on trouve dans l'i-nœud d'un fichier dans les SF traditionnels. Cette cartographie est représentée par une table d'adresses des blocs disques du fichier qui dispose d'un pointeur direct sur les premiers blocs de données, et d'une série de pointeurs d'indirection pour retrouver le reste des blocs. Le SF de PAROS utilise pour chaque disque du système, un tableau indexé par les UIDs des fichiers présents dans le disque, qui donne accès à la table d'adresses des blocs. La figure 7.3 montre la structure de ce tableau ; les numéros à l'intérieur des tables d'adresses représentent des blocs physiques disque.

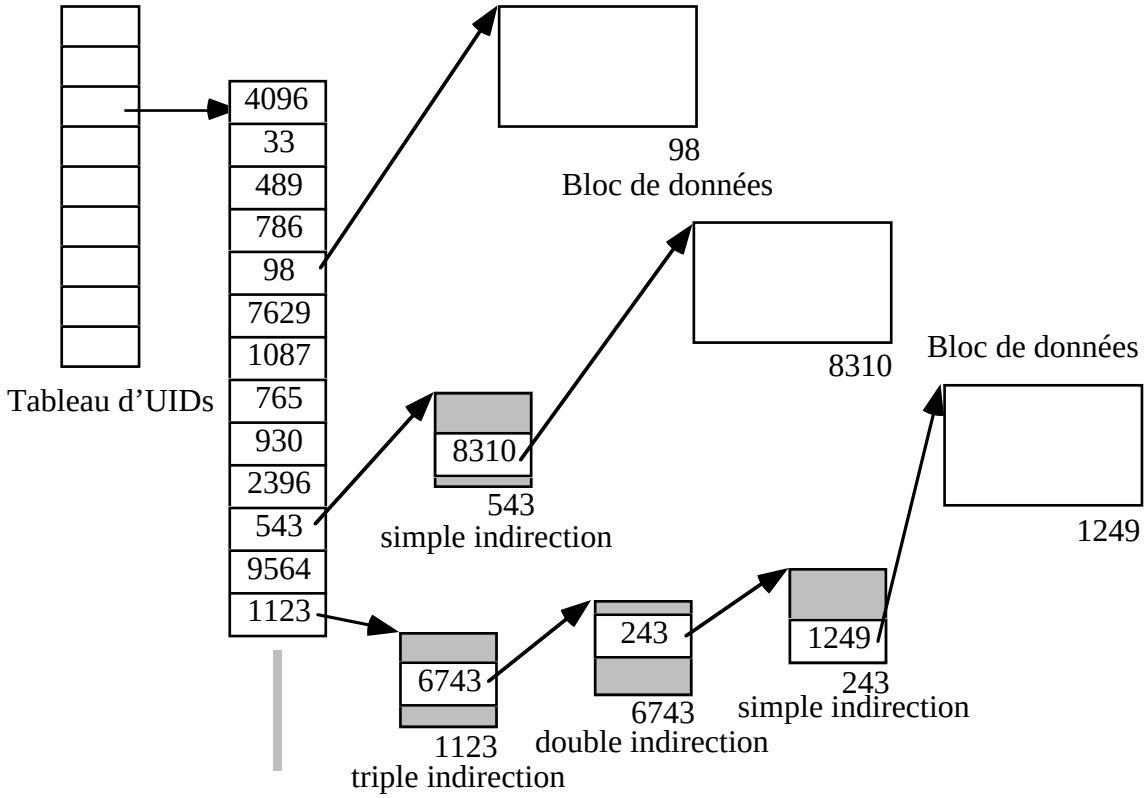


Figure 7.3 Tableau d'adresses des blocs disques des fichiers.

7.2.2 Architecture logicielle

Alors que le principe de maintient de la localité nous suggère de placer les serveurs du SF aussi près que possible de leurs processus clients, le principe d'exploitation du parallélisme nous incite à distribuer au maximum ces serveurs sur toute l'architecture. Ce problème a déjà été rencontré lors de la conception de ParX et la solution retenue est celle de la gestion d'une machine parallèle par le système des clusters utilisateurs. En effet, un cluster délimite le domaine de communication d'une Ptâche, renforçant ainsi la localité de son exécution, et l'existence des Ptâches système, et notamment de la Ptâche de contrôle, permet d'assurer les services systèmes

en parallèle pour toutes les applications. Le SF de PAROS base alors son architecture sur ce système de clusters.

La figure 7.4 présente la structure du SF de PAROS. Trois types de serveurs sont utilisés : des serveurs locaux aux nœuds (LFS pour *Local File Servers*) qui s'exécutent sur chaque nœud du cluster, des serveurs locaux aux clusters ou serveurs régionaux (RFS pour *Regional File Servers*) qui s'exécutent sur les nœuds de contrôle du cluster, et des serveurs globaux à toutes les ligues (GFS pour *Global File Servers*) qui s'exécutent sur tous les processeurs gestionnaires de disques de la Ptâche de gestion des périphériques dans le cluster système (processeurs d'E/S). Dans la suite nous détaillons le fonctionnement de chacun de ces serveurs.

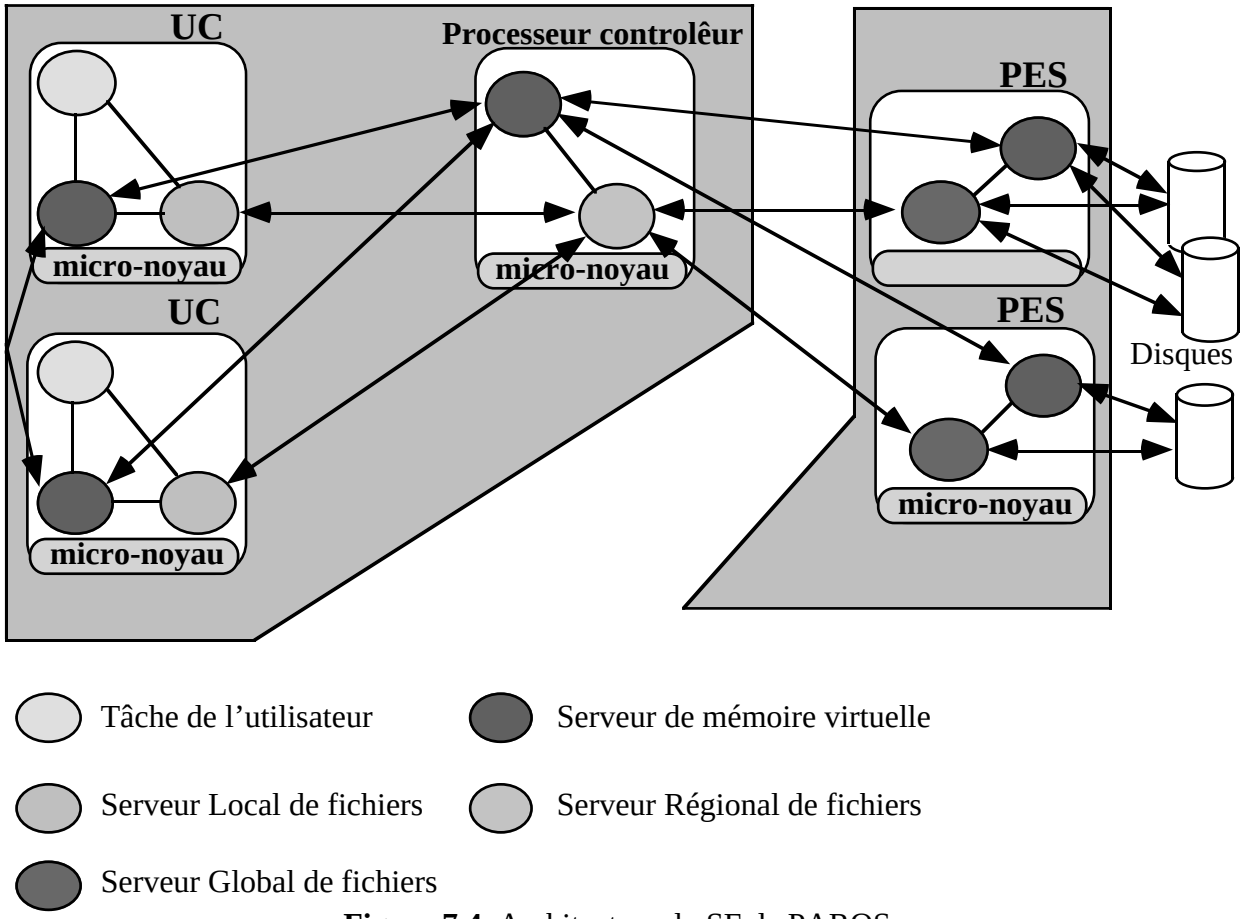


Figure 7.4 Architecture du SF de PAROS.

7.2.2.1 Serveurs de fichiers locaux (LFS)

Un serveur de fichier local s'exécute sur chaque nœud du cluster utilisateur. Les appels système de gestion de fichiers générés par les threads qui s'exécutent sur un nœud sont transformés en messages au LFS du nœud. Si l'appel est bloquant, seulement le thread qui génère l'appel est bloqué, les autres threads de la même tâche, ou des autres tâches s'exécutant sur le même nœud,

continuent normalement leur exécution. Une fois que le serveur LFS a résolu la requête (localement ou avec le concours d'autres serveurs), il réveille le thread bloqué et lui transmet le résultat de son opération.

Le rôle principal des serveurs locaux est de transformer les requêtes d'accès à un fichier en messages vers le serveur local du gestionnaire de mémoire virtuelle ou vers les serveurs régionaux du cluster. La première fois qu'un des threads du nœud accède à un fichier (opération d'ouverture), le serveur LFS fait suivre la requête vers le serveur régional et reste en attente de la réponse. Pour les fichiers déjà ouverts dans un nœud, le serveur LFS répond aux requêtes d'ouverture ultérieures d'après les informations obtenues lors de la résolution de la première requête d'ouverture des fichiers.

Pour cela, LFS conserve, pour chaque tâche qui s'exécute dans le nœud, le répertoire courant (*working directory*) qui fait partie de l'environnement de la tâche, et un tableau `Task_File_Table`, où se trouvent les informations associées aux fichiers ouverts par celle-ci. Le répertoire courant d'une tâche est le répertoire initial de recherche des fichiers dont le nom spécifié lors d'une opération d'ouverture ne commence pas par le caractère "barre oblique" ("/"). Afin de faciliter la traduction du nom symbolique en identificateur unique chez le serveur régional, LFS maintient l'UID du répertoire courant de chaque tâche et joint cet identificateur aux requêtes d'ouverture envoyées vers le serveur RFS. L'utilisation de noms de fichiers relatifs au répertoire courant est alors encouragée car celle-ci diminue le risque de congestion dans les répertoires des premiers niveaux de la hiérarchie de fichiers et, en raison de notre conception, leur traduction en UID commence directement dans le serveur LFS.

Une fois l'accès autorisé, et de ce fait le fichier projeté sur l'espace virtuel du cluster, le serveur LFS rend à la tâche à laquelle appartient le thread à l'origine de la requête, un descripteur de fichier qui correspond à l'entrée du fichier dans sa `Task_File_Table`. Ce descripteur est une variable locale à la tâche, et est utilisé par les threads de la tâche pour toutes les opérations ultérieures sur le fichier. Tous les threads d'une tâche partagent les descripteurs de fichiers, mais deux tâches qui s'exécutent sur le même nœud du cluster ont des descripteurs différents car chacune a sa propre `Task_File_Table`.

Chaque entrée du tableau `Task_File_Table` d'une tâche donnée contient les informations suivantes pour chaque fichier accédé par la tâche :

`nom`	Nom symbolique absolu du fichier
`accès`	Mode d'accès au fichier
`projection`	Adresse dans l'espace virtuel de la projection du fichier
`offset`	Adresse du pointeur de déplacement sur le fichier

Task_File_Table ne maintient pas directement la valeur du déplacement de la tâche sur le fichier car ce pointeur peut être global pour implanter un accès coopératif entre plusieurs tâches. Dans ce cas, il est implémenté comme une variable globale dans l'espace virtuel du cluster avec un compteur associé pour conserver le nombre de tâches qui partagent le pointeur. Pour les pointeurs non partagés, le pointeur de déplacement est local à la tâche.

La figure 7.5 détaille les opérations exécutées par le serveur LFS lorsqu'il arrive une requête d'ouverture d'un fichier. Pour ouvrir un fichier une tâche doit spécifier le nom symbolique du fichier (absolu ou relatif), le mode d'accès souhaité (lecture, écriture, pointeur partagé, etc.). Pour une opération de `multiopen`, la tâche peut aussi spécifier la liste de tâches qui participent à l'opération. Le serveur LFS récupère l'identification de la tâche (`id_tâche`) qui demande l'ouverture d'un fichier à partir des informations maintenues par le micro-noyau.

```
LFS_open (nom, mode, liste_tâches)

Si nom commence par "/" Alors
   nom_fic ← nom
Sinon
   nom_fic ← répertoire courant + nom
FinSi
Chercher nom_fic dans toutes les Task_File_Table locales au nœud
Si ∃ Task_File_Table / nom_fic ∈ Task_File_Table Alors
   /* Le fichier a été déjà ouvert dans ce nœud, soit tab[fd] l'entrée trouvée */
   Si mode n'est pas compatible avec tab[fd].accès Alors
      retourner (ERREUR)
   Sinon
      Si tab est la Task_File_Table associée à id_tâche Alors
         /* Le fichier était déjà ouvert par cette tâche */
         retourner (fd)
      FinSi
Sinon
   Envoyer (Serveur RFS, Requête open nom_fic mode)
   Recevoir (Réponse dans tab[fd])
FinSi
/* les données de nom_fic sont dans tab[fd] */
Prendre une entrée pour nom_fic dans la Task_File_Table de la
tâche id_tâche
/* Soit TFT[res] cette entrée */
TFT[res].nom ←nom_fic
TFT[res].accès ← mode
TFT[res].projection ←tab[fd].projection
Créer une variable locale VarP
Si pointeur_absolu ∈ mode Alors
   VarP.offset ←TFT[res].projection
Sinon
   /* Chaque tâche manipule des adresses relatives (Cf. Segmentation de fichiers)*/
```

```
   VarP.offset ←0
FinSi
 TFT[res].offset ←@VarP
Si pointeur_partagé ∈ mode Alors
   Créer une variable globale VarP
   VarP.offset ←TFT[res].projection
   VarP.cpt ← nb_tâches dans liste_tâches
   TFT[res].offset = @VarP
   bc_send (liste_tâches, TFT[res])
   retourner (res)
FinSi
```

Figure 7.5 Opération d'ouverture d'un fichier chez le serveur LFS.

`bc_send` est une primitive de diffusion sélective du micro-noyau ParX qui permet l'envoi d'un message à un groupe de tâches de façon synchrone ou asynchrone [DM93]. La réception de ces messages peut être faite par l'utilisateur ou par le serveur LFS (opération `bc_receive`).

Les opérations de lecture et d'écriture sur un fichier sont transformées par le serveur LFS en requêtes de lecture et d'écriture sur l'espace virtuel d'après l'adresse de la projection du fichier dans l'espace virtuel et le déplacement de la tâche sur le fichier (*offset*). Ces requêtes sont traitées par le gestionnaire de mémoire virtuelle du nœud comme il est expliqué dans le chapitre 5, et une fois les pages correspondantes sont chargées dans la mémoire locale du nœud, le gestionnaire rend le contrôle au serveur LFS pour que celui-ci fasse la copie des données entre les variables de l'application et la mémoire locale. La cohérence des accès concurrents est alors garantie par le gestionnaire de mémoire virtuelle.

L'algorithme des opérations de lecture et d'écriture dans un serveur LFS est présenté dans la figure 7.6. Les paramètres utilisés sont l'entrée dans la Task_File_Table, le nombre d'octets à lire ou à écrire et le tampon de l'application. Si le fichier est traité par segments, la tâche doit aussi spécifier l'adresse du segment en cours de traitement.

```
LFS_read/LFS_write (fd, nb_octets, tampon, ad_segment)

/* Soit TFT la Task_File_Table de la tâche qui exécute l'accès */
/* ad_segment=0 signifie que TFT[fd].offset↑.offset est un pointer absolu */
ad_virtuelle ← ad_segment + TFT[fd].offset↑.offset
Si opération de LECTURE Alors
   oct ← copy_mem (tampon, ad_virtuelle, nb_octets)
Sinon
   oct ←copy_mem (ad_virtuelle, tampon, nb_octets)
FinSi
TFT[fd].offset↑.offset ← TFT[fd].offset↑.offset + oct
```

```
retourner (oct)
```

Figure 7.6 Opérations de lecture/écriture.

`copy_mem` est une opération de copie mémoire entre l'espace virtuel et une zone de la mémoire locale ; cette opération fait appel au gestionnaire de mémoire virtuelle. La détection du caractère EOF (**E**nd **O**f **F**ile) pendant la lecture termine l'opération et retourne le nombre d'octets effectivement lus.

De la même façon que le système cherche à traiter localement les opérations d'ouverture d'un fichier, les serveurs LFS essaient de résoudre les requêtes de fermeture des fichiers sans s'adresser aux serveurs régionaux. Pour cela, la requête de fermeture d'un fichier est envoyée au serveur local qui a traité la requête d'ouverture du fichier, et celui-ci ne la transmet au serveur régional que lorsqu'il est sûr que toutes les tâches auxquelles il a donné accès au fichier ont exécuté une opération de fermeture sur le fichier.

La figure 7.7 présente l'algorithme de l'opération de fermeture d'un fichier. Si le fichier est traité par segments, la tâche doit aussi spécifier l'adresse du segment traité. La mise à jour du fichier sur disque est réalisée par le gestionnaire de mémoire virtuelle en collaboration avec le serveur régional de fichiers.

```
LFS_close (fd, ad_segment)

Si ad_segment = 0 Alors
   /* Fichier accédé en entier */
   Mise à jour du fichier sur disque (opération de flush)
Sinon
   Mise à jour du segment du fichier sur disque
FinSi
/* Soit TFT la Task_File_Table de la tâche qui ferme le fichier */
Si pointeur_partagé ∈ TFT[fd].accès Alors
   TFT[fd].offset↑.cpt --
   Si TFT[fd].offset↑.cpt = 0 Alors
      Envoyer (Serveur RFS, Requête close TFT[fd].nom)
   FinSi
Sinon
   Envoyer (Serveur RFS, Requête close TFT[fd].nom)
FinSi
```

Figure 7.7 Opération de fermeture d'un fichier chez le serveur LFS.

7.2.2.2 Serveurs de fichiers régionaux (RFS)

Les serveurs de fichiers régionaux ont à leur charge le traitement des requêtes d'ouverture et de fermeture de fichiers. C'est le serveur RFS qui commence la traduction d'un nom symbolique en l'identificateur unique du fichier. Pour ce faire, il maintient une copie du répertoire racine avec la liste des UIDs pour tous les fichiers dans ce répertoire. Dans le cas des requêtes d'ouverture qui spécifient un fichier à partir de leur nom absolu (nom commençant par le caractère "/"), ceci permet de distribuer plus uniformément les requêtes d'ouverture sur les différents disques que si l'on conserve seulement l'UID du répertoire racine. Un protocole de diffusion sur la Ptâche de contrôle est utilisé pour maintenir à jour les copies du répertoire racine des serveurs RFS sur les différents clusters, ce qui devrait être une opération peu fréquente. Pour les fichiers spécifiés par leur nom relatif (par rapport au répertoire courant de la tâche), le serveur RFS continue la traduction à partir de l'UID de ce répertoire qu'il reçoit des serveurs LFS avec la requête d'ouverture.

Le serveur RFS réalise la première partie de la traduction du nom symbolique et obtient l'identification du disque sur lequel celle-ci doit se continuer ; il envoie ensuite au serveur global qui s'exécute sur le PES qui est chargé de la gestion de ce disque. Une fois la traduction finie, l'i-nœud du fichier est chargé dans le serveur RFS qui a commencé la traduction. L'utilisation de la fonction de hachage, qui à partir du nom symbolique calcule l'unité de disque où stocker l'i-nœud du fichier, est exclue car le résultat d'une telle fonction varie selon le nombre de disques du système. Ainsi, les i-nœuds des fichiers dont les dernières composantes des noms symboliques sont identiques peuvent se retrouver sur des disques différents si le calcul de la fonction de hachage est fait avec des configurations distinctes du sous-système d'E/S.

C'est alors le serveur RFS qui réalise la vérification des droits d'accès d'une Ptâche (celle à laquelle appartient la tâche à l'origine de la requête) à un fichier. Si la Ptâche est autorisée à accéder le fichier, le serveur RFS alloue dans la mémoire virtuelle une partition pour la projection du fichier et envoie l'adresse de cette partition au serveur LFS qui a demandé l'ouverture du fichier. Si au contraire le fichier ne peut pas être accédé par la Ptâche avec le type d'accès demandé, un message d'erreur est renvoyé vers le serveur LFS.

Un fichier n'est projeté qu'une fois sur l'espace virtuel. Un tableau, `League_File_Table`, sert au serveur régional à garder une trace de tous les fichiers ouverts par la ligue. Ce tableau lui permet, entre autres, de refuser l'ouverture d'un fichier en écriture si le fichier en question est déjà accédé en lecture dans cette ligue, ou de refuser tout accès si le fichier est ouvert en écriture. Si un fichier a été ouvert en lecture et qu'une deuxième requête d'ouverture en lecture du fichier

arrive au serveur régional, celui-ci peut la résoudre localement grâce aux informations conservées dans le tableau League_File_Table.

Pour chaque fichier ouvert dans le cluster, les serveurs RFS maintiennent dans League_File_Table les informations suivantes :

nom	Nom symbolique absolu du fichier
accès	Mode d'accès au fichier
cpt	Nombre de requêtes d'ouverture résolues au niveau régional
i-nœud	Pointeur vers l'i-nœud du fichier
uid	UID du fichier
projection	Adresse dans l'espace virtuel de la projection du fichier

La figure 7.8 présente l'algorithme exécuté par le serveur RFS lors de la réception d'une requête d'ouverture de fichier en provenance d'un serveur local. Autre le nom du fichier et le mode d'accès, la requête contient l'UID du répertoire courant de la tâche qui exécute l'ouverture, l'identification de la Ptâche à laquelle appartient cette tâche et l'identification du serveur LFS qui envoie la requête.

```
RFS_open (nom, mode, UIDrep, id_Ptâche, id_LFS)

Chercher nom dans League_File_Table
Si nom ∉ League_File_Table Alors
   Si UIDrep = 0 Alors
      varUID ←UID de la première composante du nom symbolique
   Sinon
      varUID ← UIDrep
   FinSi
   UIDfic ← traduction (nom, varUID)
   Vérification des droits de la Ptâche id_Ptâche (mode)
   Prendre entrée pour nom dans League_File_Table
   /* Soit L_F_T[res] cette entrée */
   L_F_T[res].nom ← nom
   L_F_T[res].cpt ← 0
   L_F_T[res].accès ← mode
   L_F_T[res].uid ← UIDfic
   /* Chargement de l'i-nœud du fichier */
   gfs ← Gestionnaire du disque UIDfic.disque
   Envoyer (gfs, Requête chargement bloc 0)
   Recevoir (Vari-nœud)
   L_F_T[res].i-nœud ←@Vari-nœud
   /* Projection du fichier sur l'espace virtuel */
   L_F_T[res].projection ← malloc (Vari-nœud.taille)
FinSi
/* Soit res l'entrée de nom dans League_File_Table */
Si mode n'est pas compatible avec L_F_T[res].accès Alors
```

```
   retourner (ERREUR)
FinSi
L_F_T[res].cpt++
Envoyer (Serveur LFS, L_F_T[res].projection)
```

Figure 7.8 Opération d'ouverture d'un fichier chez le serveur RFS.

Le nombre d'opérations de fermeture d'un fichier qui arrivent à un serveur RFS doit être égal à celui des opérations d'ouverture du fichier traitées par le serveur. Pour assurer cet équilibre, nous avons vu que les serveurs LFS qui résolvent des requêtes d'ouverture, traitent eux même les opérations de fermeture et ne préviennent les serveurs RFS que lorsqu'ils sont sûrs que toutes leurs tâches ont fermé le fichier. Pour les requêtes qui arrivent au serveur RFS, celui-ci conserve dans `League_File_Table` un compteur qui lui permet de déterminer le moment où un fichier n'est plus accédé par une tâche de la ligue. Une fois la valeur de ce compteur remise à zéro, le serveur RFS détruit la projection du fichier dans l'espace virtuel du cluster. La figure 7.9 détaille l'algorithme de l'opération de fermeture dans un serveur régional. Cette primitive reçoit du serveur LFS le nom symbolique du fichier à fermer.

```
RFS_close (nom)

/* Soit L_F_T[res].nom = nom */
L_F_T[res].cpt --
Si L_F_T[res].cpt = 0 Alors
   free_mem (L_F_T[res].projection)
FinSi
Libérer l'entrée res dans League_File_Table
```

Figure 7.9 Opération de fermeture d'un fichier chez le serveur RFS.

La deuxième fonction des serveurs régionaux est de recevoir les requêtes des défauts de pages du gestionnaire mémoire du cluster. Si un défaut de page, d'une page dans la projection en mémoire d'un fichier, ne peut pas être résolu à l'intérieur du cluster, le serveur mémoire du niveau 3 (cluster mémoire de l'utilisateur, cf. section 6.3.1) ne continue pas la recherche vers le niveau suivant mais, sachant qu'il s'agit d'une page fichier[1], il transmet la requête au serveur RFS. Grâce aux informations dans le tableau `League_File_Table` (et notamment dans l'i-nœud du fichier), ce serveur est capable de déterminer les blocs disque qui contiennent les pages recherchées et de générer les requêtes nécessaires pour que les serveurs globaux chargent ces pages.

[1] Lors de la projection d'un fichier en mémoire, le gestionnaire de mémoire marque toutes les pages de l'espace virtuel utilisé par la projection comme étant des pages fichier.

L'algorithme des opérations de lecture et d'écriture dans un serveur LFS est présenté dans la figure 7.10. Le serveur VSM envoie au serveur RFS une requête pour chaque page qui doit être chargée à partir des disques.

```
RFS_read/RFS_write (adresse)

Soit res tel que adresse ∈ [begin_fic, end_fic] avec
   begin_fic ← L_F_T[res].projection
   end_fic  ←  L_F_T[res].projection  +  L_F_T[res]↑.i-
nœud.taille
offset ← adresse - L_F_T[res].projection
bloc ← offset / L_F_T[res]↑.i-nœud.taille_bloc
som ← 0
Tant que som < taille_page_virtuelle Faire
   disque_bloc ← L_F_T[res]↑.i-nœud.f_hachage (bloc)
   gfs ← Gestionnaire du disque disque_bloc
   Envoyer (gfs, Requête lecture/écriture bloc)
   Recevoir (bloc dans pag)
   som ← som + L_F_T[res]↑.i-nœud.taille_bloc
   bloc ++
FinTant que
Envoyer (Serveur VSM, page pag)
```

Figure 7.10 Opérations de lecture/écriture dans les serveurs RFS.

7.2.2.3 Serveurs de fichiers globaux (GFS)

Un serveur de fichier global s'exécute sur chaque PES de la Ptâche de gestion des disques et reçoit des requêtes de lecture/écriture de blocs disque en provenance des serveurs régionaux des toutes les ligues en exécution. Les serveurs RFS identifient le serveur global qui doit répondre à chaque requête et lui envoie l'identification du disque et le numéro du bloc du fichier dans ce disque pour effectuer l'opération d'E/S. Le serveur GFS commande ensuite l'opération d'E/S grâce à la table d'adresses des blocs disque dont il dispose pour chacun des disques à sa charge.

Aucune vérification sur les droits d'exécution des opérations n'est faite à ce niveau. En effet, toutes les requêtes reçus par les serveurs GFS proviennent des serveurs régionaux où les vérifications des droits d'accès d'une Ptâche à un fichier sont effectuées. Réaliser une vérification à ce niveau, tel qui est fait dans le SF Vesta de la section 4.2.4.1, est très pénalisant car il faudrait gérer au niveau de la Ptâche système une trace de tous les fichiers ouverts par toutes les Ptâches des utilisateurs, et adresser toutes les requêtes à un serveur de vérification avant d'effectuer l'opération d'E/S.

Du fait que les serveurs RFS envoient chaque requête au serveur global qui est en mesure d'y répondre, les serveurs GFS n'ont pas besoin de communiquer entre eux, chacun interagit seulement avec les serveurs RFS. De plus, comme les transferts demandés à un serveur GFS ne concernent que les disques qui sont à sa charge, chaque serveur GFS ne commande que les contrôleurs de ses disques. L'utilisation du réseau de contrôleurs n'est alors pas totale mais son existence est néanmoins justifiée par la flexibilité obtenue dans la configuration du sous-système d'E/S. En effet, l'ensemble de disques gérés par chaque PES peut être choisi au démarrage du système.

7.2.3 Gestion de caches et préchargement

La gestion de caches et le préchargement des données de fichiers sont réalisés au niveau régional (cluster utilisateur) et au niveau global (cluster système). L'existence d'un serveur de cache et de préchargement au même niveau que les serveurs RFS permet l'implémentation de politiques spécifiques aux applications de l'utilisateur, tandis que le serveur de cache et préchargement au niveau des serveurs GFS définit les politiques globales pour tous les clusters.

A l'intérieur d'un cluster utilisateur, le gestionnaire de mémoire virtuelle applique une politique de cache et de préchargement basée sur la notion de page qui est l'unité de gestion. Le serveur de mémoire de chaque nœud d'un cluster exécute sa propre politique de cache et de préchargement en fonction des accès à l'espace virtuel dans le nœud. De la même façon, le gestionnaire de mémoire du cluster peut décider la création d'un cluster mémoire pour limiter le nombre de défaut de pages qui ne sont pas résolues à l'intérieur du domaine de l'utilisateur.

La gestion des caches et le préchargement des données des fichiers sont donc faits en partie par le gestionnaire mémoire car les fichiers sont projetés sur l'espace virtuel lors de leur ouverture. Le SF dispose néanmoins d'un serveur propre de cache et préchargement qui s'exécute sur les processeurs de contrôle du cluster pour affiner ces fonctionnalités pour le cas des fichiers. Ce serveur coopère avec le gestionnaire de mémoire pour maintenir le taux d'occupation de la mémoire réelle du cluster entre deux seuils définis à la création du cluster.

Comme nous l'avons présenté dans la section sur la hiérarchie de la mémoire (cf. section 6.3.1) un serveur de la Ptâche système gère les tampons système où les blocs disques sont stockés temporairement afin de limiter les accès aux unités externes. Le serveur de caches et de préchargement des blocs de fichiers placé à ce niveau manipule des blocs logiques et non pas des blocs physiques, ce que lui permet de définir ses propres politiques pour maintenir dans ces tampons des données des fichiers qui risquent d'être accédées dans un futur proche.

7.3 Conclusion

Dans ce chapitre nous avons développé la conception d'un SF universel pour les architectures massivement parallèles. Pour ce faire, nous avons tout d'abord identifié trois caractéristiques fondamentales qui doivent être respectées pour une telle conception : l'extensibilité du SF, l'extension des interfaces standards et l'exploitation du parallélisme à tous les niveaux du SF. Ces caractéristiques composent les lignes directrices pour l'organisation logicielle du SF proposé.

La conception de ParX au sein de notre équipe nous a permis de baser notre SF sur un micro-noyau concret et faciliter ainsi la mise en œuvre de nos idées. Nous considérons que les caractéristiques clés de ParX qui sont utilisées par notre SF devraient être intégrées à tout micro-noyau universel pour les architectures parallèles. Malheureusement le prototype actuel de ParX ne dispose pas de toutes les fonctionnalités décrites lors de sa présentation, ce qui nous a empêché de réaliser une implémentation complète de notre système : à l'heure actuelle le modèle de processus ne reconnaît qu'un seul type de processus et la machine virtuelle à mémoire partagée n'a pas pu être intégrée à ParX parce que le prototype a été développé sur des processeurs sans unité de gestion matérielle de la mémoire (MMU).

L'architecture logicielle du SF proposé fait de la distribution des données une règle qui s'applique aussi bien aux données des fichiers des utilisateurs qu'aux structures internes de gestion du système. Ainsi, il n'existe pas de services qui peuvent devenir de goulot d'étranglement du système et l'architecture garde une forte indépendance vis-à-vis de l'architecture matérielle sous-jacente. Basé sur un micro-noyau extensible et générique, il tire profit de la machine virtuelle à mémoire partagée ce qui lui permet de bénéficier des services d'accès aux données distantes et de la gestion de leur cohérence. Enfin, la définition hiérarchique des serveurs de fichiers (locaux, régionaux et globaux) renforce la localité des services offerts aux différents niveaux de processus des applications parallèles (threads, tâches, Ptâches et Ligues).

Plus qu'une interface notre SF propose une plate-forme de développement d'interfaces pour être adaptée aux besoins des sous-systèmes de plus haut niveau. Les algorithmes exposés ne sont qu'un exemple d'implémentation de certaines opérations sur les fichiers. Nous croyons que c'est le concepteur d'un sous-système celui qui est le mieux placé pour définir l'interface d'accès aux fichiers nécessaire pour son environnement.

Chapitre 8
Conclusion

Jusqu'à présent les machines massivement parallèles n'ont pas encore connu l'essor escompté. Leur coût et leur complexité d'exploitation en sont les principaux obstacles. Surmonter ces difficultés requiert un véritable système d'exploitation capable de gérer efficacement et simultanément l'exécution de plusieurs applications. La gestion de fichiers compte parmi les services de base qu'un tel système doit offrir. La réalisation efficace de ce service est complexe de par la lenteur des unités physiques de stockage, mais aussi à cause de la diversité de besoins en matière d'E/S exprimée par les applications parallèles. L'architecture matérielle et logicielle du sous-système d'E/S que nous venons d'étudier rentre dans cette problématique générale.

Dans cette thèse nous avons respecté les principes qui sont à la base du projet ParX, c'est-à-dire, le support système efficace pour l'ensemble le plus large d'applications parallèles dans le contexte des architectures massivement parallèles. L'universalité, l'extensibilité et les hautes performances ont été les lignes directrices de notre travail, et elles sont présentes dans toute décision ou proposition exposée au long de ce document.

La section suivante dresse un bilan de notre contribution où nous faisons ressortir les solutions que nous avons élaborées pour le développement d'un sous-système parallèle d'E/S. A la fin du chapitre nous présentons les perspectives de ce travail de thèse.

8.1 Bilan

Tout au long de ce rapport nous avons défendu l'idée de qu'il est possible de réaliser un sous-système d'E/S universel et performant dans un environnement parallèle et extensible si l'on y fait participer toutes les couches impliquées dans un transfert entre la mémoire vive et les dispositifs de stockage. Le matériel, le système d'exploitation (le micro-noyau), le SF et même les utilisateurs doivent coopérer afin de garantir un équilibre entre les services généraux et les

fonctionnalités spécifiques. Nous avons donc analysé chacune de ces couches et établi, spécialement pour les couches logicielles, des propositions pour la conception d'un tel système.

La compréhension de l'architecture matérielle du sous-système d'E/S nous a permis de définir une première classification de ces systèmes pour les architectures parallèles et de dégager une architecture universelle et extensible bien adapté pour l'implémentation du principe des E/S parallèles. Ensuite, l'étude des unités de disques qui sont à la base des sous-système d'E/S nous a mené aux organisations RAID des vecteurs de disques qui apparaissent comme la solution la plus adéquate, du point de vue matériel, pour faire face au déséquilibre pénalisant entre la puissance de calcul et celle des E/S dans les architectures modernes. Parmi les différentes organisations RAID proposées jusqu'à aujourd'hui, nous avons trouvé que c'est le niveau 5 (intégration des données et de l'information redondante) qui répond le mieux aux critères fixés pour notre travail.

Ensuite, nous avons été amenés à apporter à ParX trois types d'extensions pour mieux l'adapter aux besoins spécifiques des sous-systèmes d'E/S. Tout d'abord dans son modèle de processus car il nous manquait un niveau d'abstraction pour établir les règles de coopération entre plusieurs Ptâches, mais surtout pour définir l'entité système à laquelle limiter la cohérence qui doit assurer le système pour les fichiers accédés de façon concurrente. Puis, dans sa gestion des ressources physiques car le maintient en mémoire des fichiers, afin de réduire les accès aux dispositifs de stockage, augmente considérablement la consommation de ces ressources. Enfin, dans ses mécanismes d'acheminement de messages car de par sa nature, le trafic généré par les E/S nécessite d'un traitement spécial afin de limiter les interférences entre les communications inter-processus proprement dites et celles des transferts de fichiers.

La mémoire virtuelle est un service système qui permet l'utilisation des mémoires locales des différents nœuds de l'architecture comme un cache de la mémoire secondaire grâce à la projection des fichiers dans l'espace d'adressage virtuel. A partir d'une hiérarchie de mémoires nous avons proposé un protocole original de recherche de pages dans cette hiérarchie qui, afin d'améliorer ses performances, exploite directement la couche de routage de ParX. De plus, ce protocole garantit la cohérence de l'espace virtuel par une gestion basée sur la page comme unité de cohérence. Ce protocole a été implémenté et intégrée au mécanisme de routage de ParX, et il doit servir de base à la réalisation d'une machine virtuelle à mémoire partagée qui est en cours de développement dans notre équipe.

Enfin, après avoir passé en revue les principaux SF parallèles, nous avons introduit ceux que nous pensons être les principes de base qui doivent être respectés pour concilier universalité et hautes performances dans la conception d'un SF parallèle et extensible. Suite à la définition de ces principes, nous avons présenté notre SF, illustré sur les concepts propres à ParX mais

néanmoins général dans son approche. Les trois types de serveurs du SF ont été implémentés dans ParX, malheureusement l'état actuel du prototype ne comporte ni la gestion des Ptâches ni celle des clusters, ce qui nous empêche de réaliser une mise en œuvre complète de notre système.

8.2 Perspectives

La première perspective de notre travail concerne la réalisation d'un prototype où toutes les fonctionnalités décrites soient présentes. Pour cela deux travaux sont nécessaires impérativement : tout d'abord la terminaison de la mise en œuvre de ParX, avec les extensions décrites dans ce rapport ; ensuite le portage du prototype résultant sur une nouvelle machine où les processeurs intègrent un mécanisme de traduction d'adresses afin de pouvoir intégrer complètement la machine à mémoire virtuelle partagée (les machines Supernode à base de transputers n'offrent pas ce mécanisme).

Un axe de recherche qui se dégage tout naturellement de notre travail est l'étude des besoins en matière d'E/S des applications parallèles d'aujourd'hui. Quelques projets de recherche se sont déjà lancés dans cette voie mais toujours en ciblant un type spécifique d'applications. La définition d'une interface standard et indépendante de toute implémentation permettrait d'avancer à grands pas dans le développement des sous-systèmes d'E/S.

Bibliographie

[Ac+86] M. Accetta et. al, “Mach: a new kernel foundation for UNIX development”, *Proceedings of the Summer 1986 USENIX conference*, Atlanta, GA, July 1986.

[AG94] G.S. Almasi, A. Gottlieb, *“Highly Parallel Computing”*, Second Edition, The Benjamin/Cummings Publishing Company Inc., 1994.

[AGHL91] B. Abali, B.D. Gavril, R.W. Hadsell, L. Lam, B. Shimamoto, “Many/370: a parallel computer prototype for I/O intensive applications”, *Proceedings of the 6th Distributed Memory Computing Conference*, Portland, Oregon, pp. 728-730, April 1991.

[Al+90] R. Alverson et al, “The Tera computer system”, *International Conference on Supercomputing 1990*, pp. 1-6, June 1990.

[Am67] G.M. Amdahl, “Validity of the single processor approach to achieving large scale computing capabilities”, *Proceedings of AFIPS Spring Joint Computer Conference*, Atlantic City, New Jersey, Vol. 30, pp. 483-485, April 1967.

[AS89] R.K. Asbury, D.S. Scott, “Fortran I/O on the iPSC/2: is there a read after write?”, *Proceedings of the 4th Conference on Hypercube Concurrent Computers and Applications*, Monterey, California, pp. 129-132, 1989.

[Ba86] M.J. Bach, *“The design of the UNIX operating system”*, Prentice-Hall, 1986.

[Ba94] A. Balaniuk, “Adaptive page replacement in the DIVA shared virtual memory parallel server”, *Actes des Journées des Jeunes Chercheurs en Architecture de machines et systèmes*, Monastir, Tunisie, pp. 223-232, Décembre 1994.

[BBBM94] M. Blaum, J. Brady, J. Bruck, J. Menon, “EVENODD: an optimal scheme for tolerating double disk failures in RAID architectures”, *Proceedings of the 21th International Symposium on Computer Architecture*, Chicago, Illinois, pp. 245-254, April 1994.

[BBK91] R. Balter, J.P. Banâtre, S. Krakowiak, *“Construction des systèmes d’exploitation répartis - Chapitre 6 : Gestion répartie de fichiers”*, INRIA, 1991.

[BCDE93] A. Balaniuk, H. Castro, R. Despons, A. Elleuch, F. Menneteau, L. Mugwaneza, T. Muntean, E-G. Talbi, Ph. Waille, “A generic multi virtual machines parallel operating system”, *Proceedings of the International Conference on Parallel Computing PARCO*, Grenoble, France, September 1993.

[BCZ91] J.K. Bennet, J.C. Carter, W. Zwaenepoel, “Munin: distributed shared memory using multi-protocol release consistency, *Lecture Notes on Computer Science 563*, pp. 56-60, July 1991.

[BGF88] A. Barak, B.A. Galler, Y. Farber, *"A holographic file system for a multicomputer with many disk nodes"*, Technical Report CSE-TR-01-88, University of Michigan, May 1988.

[BHKS91] M.G. Baker, J.H. Hartman, J.H. Kupfer, M.D. Shirriff, J.K. Ousterhout, "Measurements of a distributed file system", *Proceedings of the 13th ACM Symposium on Operating Systems Principles*, pp. 198-212, 1991.

[BM94] A. Balaniuk, T. Muntean, "Programming with shared data in parallel loosely coupled machines: the shared virtual memory approach", *Proceedings of the International Workshop on High Perfromance Computing*, pp. 129-142, March 1994.

[BS89] B. Braban, P. Schlenk, "A well strucutred parallel file system for PM", *Operating Systems Review, ACM SIGOPS*, Vol. 23, No 2, pp. 25-38, April 1989.

[Ca91] H. Castro, "*Gestion de ressources virtuelles partagées dans les machines parallèles"*, D.E.A d'informatique à l'INPG, Grenoble, France, Septembre 1991.

[Ca93] H. Castro, "A virtual shared memory machine for PAROS", *Actes des Journées des Jeunes Chercheurs en Architecture à mémoire virtuellement partagée*, Toulouse, France, Octobre 1993.

[Ca94] H. Castro, "Vers les Entrées/Sorties à hautes performances", *Actes des Journées des Jeunes Chercheurs en Architecture de machines et systèmes*, Monastir, Tunisie, pp. 169-178, Décembre 1994.

[CBF93] P.R. Corbett, S.J. Baylor, D.G. Feitelson, "Overview of the Vesta parallel file system", *Workshop on I/O in Parallel Computer Systems*, Newport Beach, California, April 1993.

[CBHK94] A. Choudhary, R. Bordawekar, M. Harry, R. Krishnaiyer, R. Ponnusamy, T. Singh, R. Thakur, *"PASSION: Parallel And Scalable Software for Input-Output"*, NPAC Technical Report SCCS-636, Syracuse University, Syracuse, NY, September 1994.

[CEMW93] H. Castro, A. Elleuch, T. Muntean, P.Waille, "Generic microkernel architecture for the PAROS PARallel Operating System", *World Transputer Congress - Transputer Application and Systems*, R. Grebe et al., IOS Press (Eds), Vol. 2, 1993.

[CFKA90] D. Chaiken, C. Fields, K. Kurihara, A. Agarwal, "Directory-based cache coherence in large-scale multiprocessors", *IEEE Computer*, pp. 49-58, June 1990.

[CGK90] I. Chlamtac, A. Ganz, M.G. Kienzle, "*Performance evaluation of an HIPPI interconnection system"*, Technical Report 16482, IBM US Research Centers, York Town, San José, Ocober 1990.

[Ch87] *"Chrisalys programmers manual"*, BBN Advanced Computers Inc., April 1987.

[Ch88] D.R. Cheriton, "The V distributed system", Communications of ACM, Vol. 31, No 3, pp. 314-333, March 1988.

[CK93] T.H. Cormen, D. Kotz, *"Integrating theory and practice in parallel file system"*, Technical report PCS-TR93--188, Dartmouth College, March 1993.

[CL91] L.F. Cabrera, D.E. Long, “Swift: using distributed disk striping to provide high I/O data rates”, *Computing Systems*, Vol. 4, No 4, 1991.

[CLGK94] P.M. Chen, E.K. Lee, G.A. Gibson, R.H. Katz, D.A. Patterson, “RAID: high-performance, reliable secondary storage”, *ACM Computing Surveys*, Vol. 26, No 2, June 1994.

[CLVW94] P. Cao, S.B. Lim, S. Venkataraman, J. Wilkes, “The tickerTAIP parallel RAID architectures”, *Proceedings of the 20th International Symposium on Computer Architecture*, pp. 52-63, April 1994.

[CM88] L.W. Tucker, G.G. Robertson, “Architecture and applications of the Connection Machine”, *Computer*, Vol. 21, pp. 26-38, August 1988.

[CM93] H. Castro, T. Muntean, “PAROS: a parallel operating system for distributed memory parallel computers”, *Rencontres de parallélisme*, Brest, France, Juin 1993.

[Co93] F. Collin, *“Les systèmes de gestion de fichiers parallèles dédiés aux applications scientifiques : concepts, état de l’art, et analyse des applications cibles”*, DEA de l’université de Nancy I, septembre 1993.

[Cr89] T. Crockett, “File concepts for parallel I/O”, *Proceedings of Supercomputing’89*, pp. 574-579, 1989.

[CW87] L.F. Cabrera, J. Wyllie, *“QuieckSilver distributed file services: an architecture for horizontal growth”*, Technical Report RJ 5578 (56697), IBM Almaden Research Center, April 1987.

[CW93] S.S. Coleman, R.W. Watson, “New architectures to reduce I/O bottlenecks in high-performance systems”, *Proceedings of the 26th Hawaii International Conf on System Sciences*, Vol. II, pp. 31-39, January 1993.

[DBC93] J.M. Del Rosario, R. Bordawekar, A. Choudhary, “Improved parallel I/O via two-phase run-time access strategy”, *Workshop on I/O in Parallel Computer Systems*, Newport Beach, California, April 1993.

[DD93] E.P. DeBenedicitis, J.M. Del Rosario, “Modular Scalable I/O”, *Journal of Parallel and Distributed Computing*, Vol. 17, No 1,2, pp. 122-128, January/February 1993.

[De72] P.J. Denning, “On modeling program behavior”, *Proceedings of the Spring Joint Computer Conference*, Atlantic City, NJ, pp. 937-944, May 1972.

[DHC94] J.M. Del Rosario, M. Harry, A. Choudhary, “The design of VIP-FS: A virtual parallel file system for high performance parallel and distributed computing”, NPAC Technical Report SCCS-628, Syracuse University, Syracuse, NY, May 1994.

[Di90] P.C. Dibble, *“A parallel interleaved file system”*, Ph.D. dissertation, Computer Science Departament, University of Rochester, Available as technical report TR 334, March 1990.

[DJT94] D. Durand, R. Jain, D. Tseytlin, “Distributed scheduling algorithms to improve the performance of parallel data transfers”, *Workshop on I/O in Parallel Computer Systems*, Cancun, Mexico, pp. 85-104, April 1994.

[DM91] E. DeBenedicitis, P. Madams, “nCUBES’s parallel I/O with Unix compatibility”, *Proceedings of the 6th Distributed Memory Computing Conference,* Portland, Oregon, pp. 270-277, April 1991.

[DM93] R. Despons, T. Muntean, “Constructing correct protocols for a diffusion vitual machine in message passing parallel architectures”, Proceedings of the World Transputer Congress, Aachen, Germany, September 1993.

[DS89] P.C. Dibble, M.L. Scott, “Beyond striping: the Bridge multiprocessor file system”, *Computer Architecture News,* Vol. 19, No. 5, September 1989.

[DSE88] P.C. Dibble, M.L. Scott, C.S. Ellis, “Bridge: a high-preformance file system for parallel processors”, *Proceedings of the 8th International Conference on Distributed Computing Systems,* San Jose, California, pp. 154-161, June 1988.

[Ec91] H. Eckardt, *“Investigation of distributed disk I/O concepts”*, PUMA task 2.3, Working Paper, May 1991.

[El94] A. Elleuch, *“Migration de processus dans les systèmes massivement parllèles”*, Thèse de doctorat de l’Institut National Polytechnique de Grenoble, France, Novembre 1994.

[FGB91] A. Forin, D. Golub, B. Bershad, “An I/O system for Mach”, *Proceedings of the Usenix Mach Symposium,* November 1991.

[FPD93] J.C. French, T.W. Pratt, M. Das, “Performance Measurement of the Concurrent File System of the intel iPSC/2 hypercube”, *Journal of Parallel and Distributed Computing,* Vol. 17, No 1,2, pp. 115-121, January/February 1993.

[Fr87] P.D. Frank, *“Advances in head technology”*, Presentation at Challenges in Disk Technology Short Course, Institute for Information Storage Technology, University of Santa Clara, California, 1987.

[GDS86] R.F. Gurwitz, M.A. Dean, R.E. Schantz, “Programming support in the Cronus distributed operating system”, *Proceedings of the Sixth International Conference on Distributed Computing Systems,* pp. 486-493, May 1986.

[GHMP90] R.G. Guy, J.S. Heidemann, W. Mak, T.W. Page, Jr., G.J. Popek, D. Rothmeier, “Implementation of the Ficus replicated file system”, *USENIX Conference Proceedings,* pp. 63-71, June 1990.

[Gi90] M. Gien, “Micro-kernel architecture key to modern operating systems design”, *Unix Review,* Vol. 8, No 11, November 1990.

[GL91] A.S. Grimshaw, E.C. Loyot, Jr., *“ELFS: object oriented extensible file systems”*, Technical Report TR-91-14, Computer Science Department, University of Virginia, July 1991.

[GNS88] D.K. Gifford, R.M. Needham, M.D. Schroeder, “The CEDAR file system”, *Communications of the ACM,* Vol. 31, No 3, pp: 228-298, March 1988.

[Go91] N. Gonzalez, *“PARX : noyau de système pour les ordinateurs massivement parallèles - Contrôle de la communication entre processus”,* Thèse de doctorat de l’Institut National Polytechnique de Grenoble, France, Décembre 1991.

[GP91] A.S. Grimshaw, J. Prem, “High performance parallel file objects”, *Proceedings of the 6th Distributed Memory Computing Conference,* Portland, Oregon, pp. 720-723, April 1991.

[GWHP93] G.R. Ganger, B.L. Worthington, R.Y. Hou, Y.N. Patt, “Disk subsystem load balancing: disk striping vs. conventional data placement”, *Proceedings of the 26th Hawaii International Conference on System Sciences,* Vol. II, pp. 40-49, January 1993.

[GWM90] A. Gupta, W. Weber, T. Mowry, “Reducing mmory and traffic requirements for scalable directory-based cache coherence schemes”, *Proceedings of the 1990 International Confernce on Parallel Processing,* Vol. 1, pp. 312-321, 1990.

[He84] G. Held, “*Data compression: techniques and applications: hardware and software considerations”*, J. Wiley and sons (Eds), 1984.

[HLH92] E. Hagersten, A. Landin, S. Haridi, “DDM - a Cache Only Memory Architecture”, *IEEE Computer*, pp. 44-54, September 1992.

[HMSC86] J.P. Hayes, T.N. Mudge, Q.F. Stout, S. Colley, J. Palmer, “Architecture of a hypercube supercomputer”, *Proceedings of the 1986 International Conference on Parallel Processing,* pp. 653-660, 1986.

[HNS94] M. Henderson, B. Nickless, R. Stevens, “A scalable high-performance I/O system”, *Proceedings of the 1994 Scalable High Performance Computing Conference SHPCC’94,* Knoxville, TN, May 1994.

[Ho85] C.A.R. Hoare, “*Communicating sequential processes”*, Prentice-Hall, Englewoods Cliffs, 1985.

[HO93] J.H. Hartman, J.K. Ousterhout, “The Zebra striped network file system”, *Proceedings of the 14th ACM Symposium on Operating System Interface for Computer Environments,* 1993.

[In89] *“Concurrent I/O application examples”*, Intel Corporation Background Information, 1989.

[INMOS91] Inmos, “*The T9000 transputer products overview manual”*, inmos Databook series, first edition, 1991.

[Ja93] D.H. Jaffe, “Architecture of a fault-tolerant RAID-5+ I/O subsystem”, *Proceedings of the 26th Hawaii International Conference on System Sciences,* Vol. II, pp. 60-69, January 1993.

[JLGS90] D.V. James, A.T. Laundrie, S. Gjessing, G.S. Sohi, “Scalable Coherent Interface”, *IEEE Computer*, pp. 74-77, June 1990.

[JSWB93] R. Jain, K. Somalwar, J. Werth, J.C. Browne, “Scheduling parallel I/O operations”, *Workshop on I/O in Parallel Computer Systems,* Newport Beach, California, April 1993.

[KE93] D. Kotz, C.S. Ellis, “Caching and writeback policies in parallel file systems”, *Journal of Parallel and Distributed Computing,* Vol. 17, No 1,2, pp. 140-145, January/February 1993.

[KGP89] R.H. Katz, G.A. Gibson, D.A. Patterson, *"Disk system architecture for high performance computing"*, Technical Report, University of California, Berkeley, csd-89-497, 1989.

[Ki86] M.Y. Kim, "Synchronized disk interleaving", *IEEE Transactions on Computers*, Vol. C-35, No 11, November 1986.

[KN94] D. Kotz, N. Nieuwejaar, "Dynamic file-access characteristics of a production parallel scientific workload", *Proceedings of Supercomputing'94*, 1994.

[Ko92] D. Kotz, *"Multiprocessor file system interface"*, Technical Report PCS-TR92-179, Department of Math and Computer Science, Darmouth College, Hanover, NH, May 1992.

[Ko94] D. Kotz, *"Disk-directed I/O for MIMD multiprocessors"*, Technical Report PCS-TR94-226, Department of Math and Computer Science, Darmouth College, Hanover, NH, July 1994.

[KOHH94] J. Kuskin, D. Ofelt, M. Heinrich, J. Heinlein, R. Simoni, K. Gharachorloo, J. Chapin, D. Nakahira, J. Baxter, M. Horowit, A. Gupta, M. Rosenblum, J. Hennessy, "The Standford FLASH multiprocessor", *Proceedings of the 21th International Symposium on Computer Architecture*, Chicago, pp. 302-313, April 1994.

[Kr88] S. Krakowiak, *"Principles of operating systems"*, MIT Press, 1988.

[KS93] O. Krieger, M. Stumm, "HFS: A flexible file system for large-scale multiprocessors", *Proceedings of the DAGS'93 symposium, Dartmouth College*, Hanover, NH, pp. 6-14, June 1993.

[KSR92] Kendall Square Research, KSR1 Technical Summary, Walthm, MA, 1992.

[KSU92] O. Krieger, M. Stumm, R. Unrau, *"Exploiting the advantages of mapped files for stream I/O"*, Proceedings of Winter USENIX, pp. 27-42, 1992.

[KSU93] O. Krieger, M. Stumm, R. Unrau, *"The Alloc Stream Facility: a redesign of application-level stream I/O"*, Technical Report CSRI-275, University of Toronto, Canada, January 1993.

[La91] Y. Langue, *"PARX : architecture de noyau de système d'exploitation parallèle"*, Thèse de doctorat de l'Institut National Polytechnique de Grenoble, France, Décembre 1991.

[Le+92] Charles E. Leiserson et al, "The network architecture of the Connection Machine CM-5", *4th Symposium on Parallel Algorithms and Architectures*, pp. 272-285, June 1992.

[LH89] K. Li, P. Hudak, "Memory coherence in shared virtual memory systems", *ACM Transactions on Computer Systems*, Vol. 7, No. 4, pp. 321-359, November 1989.

[Li88] K. Li, "Ivy: a shared virtual memory system for parallel computing", *Proceedings of the 1988 International Conference on Parallel Processing*, Vol. II, pp. 94-101, August 1988.

[LIN93] S.J. LoVerso, M. Isman, A. Nanopoulos, "*sfs:* a parallel file system for the CM-5", *Proceedings of the 1993 Summer Usenix Conference*, pp. 291-305, 1993.

[LK91] E.K. Lee, R.H. Katz, “Performance consequences of parity placement in disk array”, Proceedings of the 4th International Conference on Architecture Support for Programming Languages and operating Systems, IEEE, New York, pp. 190-199, 1991.

[LKB87] M. Livny, S. Khoshafian, H. Boral, “Multi-disk management algorithms”, *Proceedings ACM SIGMETRICS*, pp. 69-77, May 1987.

[LKBT92] W.G. Levelt, M.F. Kaashoek, H.E. Bal, A.S. Tanenbaum, “A comparison of two paradigms of distributed shared memory”, *Software-Practice & Experience*, Vol. 22, No. 11, pp. 985-1010, November 1992.

[LLGW92] D. Lenoski, J. Laudon, K. Gharachorloo, W. Weber, A. Gupta, J. Hennessy, M. Horowitz, M.S. Lam, “The Stanford Dash multiprocessor”, *IEEE Computer*, pp. 63-79, March 1992.

[Lo85] J. Lohmeyer, “Use SCSI devices for multiprocessors, smart I/O systems”, *Electronic Design News*, Vol. 30, No. 2, pp. 183-188, 1985.

[LP92] Z. Lahjomri, T. Priol, “KOAN: a shared virtual memory for the iPSC/2 hypercube”, *Lecture Notes on Computer Science 634*, pp. 442-452, September 1992.

[Men93] F. Menneteau, “ParObj : un noyau de système parallèle à objets”, Thèse de doctorat de l’Institut National Polytechnique de Grenoble, France, Octobre 1993.

[Mes93] P. Messina, “The concurrent supercomputing consortium”, *IEEE Parallel and Distributed Technology*, Vol. 1, No 1, pp. 9-16, February 1993.

[Mo93] D. Mosberger, “Memory consistency models”, *Operating Systems Review*, pp. 18-26, January 1993.

[MPI94] Message Passing Interface Forum, *“MPI: a Message-Passing Interface standard”*, May 1994.

[MPIO95] P. Corbett, D. Feitelson, Y. Hsu, J.P. Prost, M. Snir, S. Fineberg, B. Nitzberg, B. Traversat, P. Wong, *“MPI-IO: A parallel file I/O interface for MPI, Version 0.3”*, January 1995.

[MS94] S.A. Moyer, V.S. Sunderam, “A parallel I/O system for high-performance distributed computing”, *Proceedings of the IFIP WG10.3 Working Conference on Programming Environment for Massively Parallel Distributed Systems*, April 1994.

[MT86] S.J. Mullender, A.S. Tanenbaum, “The design of a capability-based distributed operating system”, *The Computer Journal*, Vol. 29, No 4, pp. 77-100, 1986.

[Mu+89] T. Muntean et al., “PARX: a parallel operating system for transputer based machines”, *Applying transputer based parallel machines*, OUG-10, A. Bakkes (Ed), Enschede, Netherlands, pp. 115-141, April 1989.

[Mu93] L. Mugwaneza, *“Contrôle des communications dans les machines parallèles à mémoire distribuée : contribution au routage automatique des messages”*, Thèse de doctorat de l’Institut National Polytechnique de Grenoble, France, Novembre 1993.

[Mu94] T. Muntean, “A generic multi virtual machines architecture for distributed parallel operating systems design”, Heterogeneous systems workshop, IPPS’95, Cancun, April 1994.

[NB89] A.L. Narasimha Reddy, P. Banerjee, “An evaluation of multiple-disk I/O systems”, *IEEE Transactions on Computers,* Vol. 38, No. 12, pp. 1680-1690, December 1989.

[NK94] N. Nieuwejaar, D. Kotz, *“A Multiprocessor Extension to the conventional file system interface”*, Technical Report PCS-TR94-230, Department of Computer Science, Darmouth College, Hanover, NH, September 1994.

[NNSI90] U. Nagashima, F. Nishimoto, T. Shibata, H. Itoh, M. Gottoh, “An improvement of I/O function for auxialiary storage: parallel I/O for a large scale supercomputing”, *Proceedings of the* International *Conference on Supercomputing,* ACM press SIGARCH, Amsterdam, pp. 48-59, June 1990.

[NS89] J.P Le Narzul, M. Shapiro, “Un service de nommage pour un système à objets répartis”, *Actes Convention Unix 89,* AFUU, pp. 73-82, Paris, mars 1989.

[OCDN88] J.K. Ousterhout, A.R. Cherenson, F. Douglis, M.N. Nelson, B.B.Welch, “The Sprite network operating system”, *IEEE Computer,* Vol. 21, No 2, pp. 23-36, February 1988.

[OCHK85] J.K. Ousterhout, H.D. Costa, D. Harrison, J. Kunze, M. Kupfer, J. Thompson, “A trace driven analysis of the UNIX 4.2 BSD file system”, *Proceedings of the 10th ACM Symposium on Operating Systems Principles,* pp. 15-24, December 1985.

[Or72] E.J. Organick, *“The Multics system: an examination of its structure”*, The MIT press, Cambridge, MA, 1972.

[Pa88] M. Papageorgiou, “Le système de gestion de fichiers dans Chorus”, *Techniques et Sciences Informatiques,* Vol. 7, No 4, mai 1988.

[PEKN95] A. Purakayashta, C.S. Ellis, D. Kotz, N. Nieuwejaar, M. Best, “Characterizing parallel file-access patterns on a large-scale multiprocessor”, *Proceedings of the 9th International Parallel Processing Symposium IPPS’95,* Santa barbara, CA, April 1995.

[PFDJ89] T.W. Pratt, J.C. French, P.M. Dickens, S.A. Janet, Jr., “A comparison of the architecture and performance of two paralllel file systems”, *Proceedings of the 4th Conference on Hypercube Concurrent Computers and Applications,* Monterey, California, pp. 161-166, 1989.

[PGK88] D.A Patterson, G. Gibson, R.H. Katz, “A case for redundant arrays of inexpensive disks (RAID)”, *Proceedings of the International Conference on Management of Data,* SIGMOD, Chicago, Illinois, June 1988.

[Pi89] P. Pierce, “A Concurrent File System for a highly parallel mass storage subsystem”, *Proceedings of the 4th Conference on Hypercube Concurrent Computers and Applications,* Monterey, California, pp. 155-160, 1989.

[PPT92] D. Presotto, K. Thompson, H. Trickey, “Plan 9, a distributed system”, *USENIX Workshop on Micro-kernels and Other Kernel Architectures,* Seattle, Wa, April 1992.

[Ro+88] M. Rozier et al., “CHORUS distributed operating system”, *Computing Systems Journal*, Vol. 1, No 4, The Usenix Association, December 1988.

[Sa85] R. Sandberg, “The Sun Network File System: design, implementation and experience”, *Proceedings of the Summer’85 USENIX workshop*, 1985.

[Sa90] M. Satyanarayanan, “Scalable, secure and highly available distributed file access”, *IEEE computer*, pp: 9-21, May 1990.

[Sc84] R.E. Schantz, “*Elementary File System”*, Technical Report, DOS-79, BBN, April 1984.

[Sc90] W. Schröder-Preikschat, “PEACE - a distributed operating system for high performance multicomputer systems”, *TLecture Notes on Computer Science N° 433 : Progress in Distributed Operating Systems and Distributed Systems Management*, W. Schröder-Preikschat, W Zimmer (eds), 1990.

[SG86] K. Salem, H. Garcia-Molina, “Disk Striping”, *IEEE 1986 International Conference on Data Engineering*, 1986.

[Su90] V.S. Sunderam, “PVM: a framework for parallel distributed computing”, *Concurrency: Practice and Experience*, Vol. 2, No 2, pp. 315-339, December 1990.

[SuII91] Supernode II Workpackage 2, *“SNOS Kernel Specifications”* ESPRIT Project 2528 - SUPERNODE II. Operating Systems and Programming Environments for Parallel Computer, June 1991.

[SUK92] M. Stumm, R. Unrau, O. Krieger, “Designing a scalable operating system for shared memory multiporcessors”, *USENIX Workshop on Micro-kernels and Other Kernel Architectures*, pp. 285-303, Seattle, Wa, April 1992.

[SZ90] M. Stumm, S. Zhou, “Algorithms implementing distributed shared memory”, *IEEE Computer*, pp. 52-64, May 1990.

[Ta87] A.S. Tanenbaum, *“Operating systems: design and implementation”*, Prentice-Hall, Englewoods Cliffs, NJ, 1987.

[Ta90] C.D. Tait, *“Techniques for building highly available distributed file systems”*, Technical Report CUSU-497-89, Departement of Computer Science, Columbia University, NY, March 1990.

[Ta93] E.G. Talbi, “Allocation de processus sur les architectures parallèles à mémoire distribuée”, Thèse de doctorat de l’Institut National Polytechnique de Grenoble, France, Mai 1993.

[TM82] A.S. Tanenbaum, S.J. Mullender, “An overview of the Amoeba distributed operating system”, *Parallel Computing and Computation*, J. van Leeuwen, J.K. Lenstra, (Eds), 1982.

[Wa91] P. Waille, *“La communication dans les multiprocesseurs à connectique programmable : réconfiguration et routage”*, Thèse de doctorat de l’Institut National Polytechnique de Grenoble, France, Septembre 1991.

[Wi91] R. N. Williams, *“Adaptive data compression”*, Kluwer academy publishers, reproduction of the author’s thesis, Adelaide, Australia, June 1989, 1991.

[WO86] B. Welch, J. Ousterhout, “Prefix tables: a simple mechanism for locating files in a distributed system”, *Proceedings of the Sixth International Conference on Distributed Computing Systems,* pp: 184-189, IEEE, May 1986.

[Wr89] M. Wright, “Fast disk data rates spur IPI acceptance”, *Electronic Design News*, Vol. 34, 1989.

[ZL77] J. Ziv, A. Lempel, “A universal algorithm for sequential data compression”, IEEE Transactions on Information Theory, Vol. 23, No. 3, pp. 337-343, 1977.

[ZRBP93] R. Zajcew, P. Roy, D. Black, C. Peak, P. Guedes, B. Kemp, J. Lo Verso, M. Leibensperger, M. Barnett, F. Rabii, D. Netterwala, “An OSF/1 Unix for massively parallel multicomputers”, *Proceedings of the Usenix Winter Conference, San Diego,* California, pp. 449-468, January 1993.