Laboratoire ID

Haute performance et entrées/sorties au sein des applications parallèles

Adrien Lebre

Mars 2004

Bull

Institut National Polytechnique de Grenoble
INPG

Haute performance et entrées/sorties au sein des applications parallèles

Adrien Lebre

{adrien.lebre}@imag.fr
Laboratoire ID-IMAG
Montbonnot St Martin, Isère

{adrien.lebre}@bull.fr
BULL R&D - Open Source Division
Echirolles, Isère

Résumé

La gestion des données est, depuis fort longtemps, une composante clé au sein d'un système informatique. Dépendants de l'architecture sous-jacente, les systèmes de fichiers n'ont cessé de s'adapter dans le but de proposer des solutions permettant d'exploiter le maximun des capacités des plate-formes sur lesquel ils sont déployés (agrégation des débits, puissance d'analyse, ressources importantes de stockage) tout en tenant compte des diverses contraintes (hétérogénéité, tolérance aux pannes, passage à l'échelle). Parallèlement aux considérations matérielles énoncées ci-dessus, le développement croissant d'applications scientifiques distribuées a modifié la conception des systèmes de fichiers modernes et a imposé la prise en compte de nouveaux paramètres (accès concurrent à grains fins, multiplicité des accès non contigus ...).

Les méthodes courantes d'accès (open, read, write, close, ...) fournies par les différents systèmes de stockage sont souvent mal adaptées à la programmation parallèle. Elles ne permettent pas, par exemple, d'accéder de manière optimale à des informations diffuses au sein d'une même ressource. L'utilisation de librairies, qui viennent compléter l'API Unix standard afin d'améliorer les performances, s'avère être la solution mise en oeuvre dans la plus part des cas. En 1997, le consortium MPI a définit une nouvelle interface dans le but de définir les divers prototypes d'entrées/sorties au sein des applications parallèles. C'est sur ce standard, communément appelé MPI I/O, que nous nous attarderons. Nous verrons les problèmes inhérents à un tel modèle et la difficulté de mettre en place une implémentation portable et performante et ceci même si plusieurs optimisations apparaissent peu à peu dans les nouveaux systèmes de fichiers.

Mots clés : Cluster, système de fichiers, Entrées/Sorties parallèles, *Parallel I/O, MPI I/O, High Performance Computing*.

Table des matières

Table des figures

Introduction

Une véritable ferveur est apparue autour des techniques visant à rendre transparente et performante l'utilisation, pour le calcul, de plusieurs ordinateurs et/ou supercalculateurs inter-connectés par un réseau local. La principale raison est que ce type d'architecture, communément appelé grappe[1] ou encore *cluster* , permet de maximiser le ratio prix/performance.

De nombreux travaux de recherches ont été et sont actuellement menés dans ce domaine. Ils tentent de répondre aux problèmes et contraintes qui sont liés à ce nouveau type d'environnement d'exécution concernant l'ordonnancement, l'algorithmique parallèle, la programmation ainsi que l'accès et le partage à distance de l'ensemble des ressources.

Profitant des capacités de calculs résultantes, un grand nombre d'applications scientifiques (climatologie, biologie moléculaire, génomique, physique des hautes énergies ou encore traitement d'images ...) a été développé. Parmi ces nouveaux systèmes de résolutions performants, de plus en plus s'appuient sur une grande quantité de données et utilisent de manière intensive le gestionnaire de données sous-jacent.

Cette implication croissante des entrées/sorties au sein des applications hautement parallèles a imposé des nouvelles contraintes ainsi que des nouveaux modèles de programmation. Deux principaux axes de recherches ressortent des différentes approches proposées par la littérature ; le but consistant, bien sur dans les deux cas, à éliminer au maximum l'ensemble des différents goulets d'étranglements susceptibles d'intervenir en cours d'exécution.
La première classe de solutions se compose des techniques d'analyse de code durant la phase de compilation ; créer un graphe de flot de données à partir des instructions offre la possibilité de mettre en place diverses techniques de placement et/ou d'optimisation et ainsi diminuer l'effet indésiré de toute latence liées aux entrées/sorties[2]. Toutefois, la complexité des codes scientifiques (irrégularité des problèmes, dépendance entre les instructions) ne permet pas dans de nombreux cas de retrouver ce graphe de flot de données de manière transparente, il est alors indispensable que le développeur indique par l'intermédiaire d'instructions explicites cette enchevêtrement d'opérations d'échanges de données.
Le second groupe, auquel nous allons nous intéresser, consiste à intervenir durant l'exécution du programme soit au niveau des systèmes de partage de données en proposant des modèles adéquates aux nouvelles exigences, soit au niveau des routines d'accès en complétant les interfaces permettant d'accéder aux unités de stockages sous-jacentes.

[1]Le laboratoire ID s'est spécialisé dans ce genre d'architecture et possède après le icluster, une nouvelle grappe à base d'Itanium.

[2]Nous utiliserons par la suite, l'acronyme *I/O* , *Input/Output* pour définir les termes d'entrée/sortie.

Ainsi, depuis plusieurs années, de nombreux systèmes de fichiers, dit "parallèles" ont fait leur apparition (Vesta[CF01], PVFS [CLRT00], NFSp[LD02] ...). Partant du fait qu'une distribution des requêtes I/O sur plusieurs disques[3] améliore les temps de réactivité et fournit donc de meilleure performance ; ces systèmes de fichiers distribuées propose une A.P.I. plus ou moins fournit de routines I/O parallèles et permettent de diminuer en partie les phénomènes de saturations. Malheureusement l'utilisation de ces interfaces spécifiques rend les programmes dépendant et par conséquent non portable. Simultanément, un certain nombre de librairies [MS95, HER$^+$95, MCFX97, Raj02, CBH$^+$94] utilisateurs ont été développés au dessus des standards usuels locaux comme XFS [**?**] ou réparties comme NFS [PJS$^+$94]. Ces nouvelles interfaces, comme nous l'avons indiqué précédemment, viennent parfaire les routines d'accès UNIX (open, close, read, write ...) usuels qui sont souvent inadaptés lors du développement d'application *HPC* [4]. Le standard *MPI I/O* qui est largement décrit tout au long de ce rapport découle de l'ensemble de ces travaux.

L'extension des grappes aux grilles [FK90] (interconnexion de plusieurs architectures de calculs par un réseau dédié comme VTHD) a fait surgir de nouveaux paramètres tels que la sécurité et l'hétérogénéité des différentes «ossatures». En effet, une application distribuée sur une grille doit accéder conjointement à plusieurs Intranets en tenant compte des multiples propriétés des réseaux, des configurations et des principales fonctionnalités de chaque sous système. De plus, le fait que certaines structures ne soient accessibles qu'à des périodes spécifiques (utilisation de réseau d'entreprise la nuit ou pendant les congés) entraîne là aussi, d'importantes complications (découverte dynamique de ressources, résilience ...) en plus d'un nouveau changement d'échelle en terme de nombre d'ordinateurs.

Méthodes d'accès locales, partages de ressources au sein d'un LAN, au sein d'un cluster ou encore échange d'informations sur le web via les nombreux systèmes pair à pair comme Napster, Gnutella ou encore Freenet ; la problématique du partage des données a toujours suscité un engouement certain. La large diversité des travaux recherches menées rend très complexe voir impossible la rédaction d'un rapport recouvrant l'ensemble des problèmes énoncés dans cette introduction. Par conséquent, l'ensemble des efforts a été dirigé vers l'étude des entrées/sorties au sein des applications parallèles. Les contraintes de passage à l'échelle, de sécurité ou encore de résilience ne seront pas traitées au sein de ce document.

Une caractérisation des applications *HPC I/O* intensive figurera dans un premier chapitre. Nous y présenterons les diverses classes de comportement, la terminologie ainsi que les méthodes élémentaires spécifique à l'ensemble des travaux de recherches sur les entrées/sorties parallèles. Par la suite, nous réaliserons une description succincte des différentes approches qui ont été réalisées au cours des dernières années. Nous nous attarderons de manière plus exhaustive sur la librairie *MPI I/O* au cours d'un troisième chapitre. Nous essayerons d'y soulever l'ensemble des limites du modèle en s'appuyant sur l'implémentation ROMIO de MPICH 1.2.5.

Enfin, nous énoncerons, durant la conclusion, les diverses perspectives entrouvertes par l'étude menée et le vif intérêt ....

[3]Ce concept est plus communément connu sous le nom de *data stripping*.

[4]*High Performance Computing*

Première partie

Etat de l’art

Chapitre 1

Notions élémentaires

L'informatique a fourni le moyen de traiter rapidement et de conserver des informations dans de nombreux domaines d'activités : industriel, économique ou social. La quantité de données générée par certains développements modernes (climatologie, biologie moléculaire, génomique, physique des hautes énergies, traitements d'images ...) s'avère gigantesque. De nombreux travaux ont révélé que les performances de tels applications parallèles sont intimement corrélées aux méthodes et aux politiques de gestion de données choisies.

Dès 1967, [amd67] a soulevé cette dépendance en précisant que la gain maximal pouvant être obtenu lors de la parallélisation d'un programme est limité par les performances du sous-système le plus «lent»[1]. Partant de ce constat, un principe de base est suggéré : une machine est équilibré lorsqu'elle possède 1Mo de mémoire vive et une bande passante *I/O* de 1Mbits/s pour une puissance *CPU* de 1 *MIPS*[2]. Ainsi en appliquant cette règle aux machines actuelles, le plus «petit» calculateur du top 500 [3] requiert une bande passante de 30Go/s alors que le plus performant nécessiterait plus de 4To/s. L'intérêt de ces propos n'est pas de remettre en cause la justesse de la loi proposée par Amdahl mais d'illustrer l'écart qu'il existe entre les performances des *CPUs* modernes et les débits fournit par les unités de stockages existantes sur le marché. Ecart rappelé plus récemment dans [HP96] ou il est indiqué que si la rapidité des processeurs augmente d'environ 35% à 50% et que les temps d'accés à la mémoire diminue aux alentours de 60% par an depuis la dernière décennie, les performances apportaient aux sous-systèmes *I/O* restent très marginales en atteignant seulement 10% de gain chaque année.

La mise en place de solutions de stockages efficaces à court et à long terme est donc prépondérante ; comprendre tous les aspects essentiels pour y parvenir est le but de ce premier chapitre. Nous allons nous attarder tout d'abord, sur le comportement des opérations d'entrée/sortie au sein des applications *HPC* . Cette étape va permettre au lecteur de mieux appréhender l'intérêt de l'agrégation des requêtes *I/O* abordée au sein d'un second paragraphe ainsi que l'usage de technique asynchrone décrite à la suite. Enfin, nous terminerons cette première partie en répertoriant les aspects primordiaux qui doivent être proposés par un système de fichiers qui se veut performant.

[1] Une application dont le temps est réparti de la manière suivante : 90% calcul et 10% *I/O* , sera toujours, quelques soit le nombre de processeurs utilisé, dépendante de la phase d'entrée/sortie.

[2] Million Instructions Par Seconde.

[3] Liste des 500 plus grosses plates-formes de calculs actuelles, `http://www.top500.org`.

1.1 Caractérisation des Entrées/Sorties

Plusieurs études ayant pour but la caractérisation des divers comportements des entrées/sorties [CACR95, NKP+96] ont permis de définir de manière exhaustive l'ensemble des besoins I/O pouvant intervenir durant l'exécution d'applications scientifiques hautement parallèle. Ainsi même si l'utilisation des entrées/sorties est généralement très distinct d'un système à un autre, nous verrons que des techniques "génériques" [TGL02, NL97] ont été mises en place suite aux diverses classifications et aux nombreuses similitudes que partage l'ensemble de ces programmes.

1.1.1 Généralités

Comme dans les systèmes locaux, le travail d'un gestionnaire de fichiers en environnement parallèle est de classer les programmes et les données pour les rendre accessibles à la demande et ce en tout point de l'architecture. L'ensemble de la littérature abordant ce sujet s'appuie sur une terminologie spécifique [Leb02] permettant de caractériser les diverses notions et techniques employées.

Partie intégrante du système d'exploitation, un système de fichiers peut être fonctionnellement divisé en plusieurs parties. Nous verrons que la "couche haute" qui fournit un ensemble de services aux clients (primitive d'ouverture, de création et d'accès aux ressources) est souvent incomplète et inadaptée à la programmation parallèle.

Parmi les paramètres utilisés pour l'évaluation des routines d'accès, le facteur de réactivité[4] s'avère être un critère fondamentale dans les *I/O* parallèles. L'emploi de plusieurs «interfaces» d'accès[5] a nettement permis l'amélioration de ce coefficient. Ainsi de nos jours, la littérature ne parle plus de systèmes de fichiers distribués mais de systèmes de fichiers dits «parallèle». Le concept principale étant l'usage de plusieurs unités de stockage (nœuds, disques et/ou interfaces réseaux) permettant la parallélisation de l'ensemble des requêtes et la diminution des temps de latence. Néanmoins, [BCR93] a révélé une limite au sein de cette approche. En effet, une étude approfondie des performances physiques de l'architecture (quantité de requêtes pouvant être générée par un nœud de calcul, débits réseaux, latence *I/O* ...) est requise afin de ne pas pénaliser l'architecture par un sur-dimmensionnement (ou un sous-dimensionnement) inadéquat et une gestion trop coûteuse.

1.1.2 Classification

Bien que l'usage des entrées / sorties au sein des applications *HPC* soit variée, trois principales classes de comportement ont été définies suite aux travaux réalisés dans le cadre de l'initiative «*Scalable I/O*», [CACR95] :

Compulsive, l'ensemble des accès est réalisé au(x) même(s) moment(s) : en début d'exécution, pour lire les paramètres entrants du problème, ou encore à la fin pour pérenniser les résultats. Il est important de proposer de fort débits et un équilibrage optimale des accès pour limiter au maximum les phénomènes de saturation ponctuels. Dans tous les cas, il paraît très difficile d'éliminer ou de réduire le temps de réactivité de ce genre d'accès par des techniques de cache

[4]Le terme de réactivité évoque la quantité de temps nécessaire pour satisfaire une requête (latence réseau, analyse de la demande et formulation de la réponse).

[5]utilisation de plusieurs disques (*RAID*) ou de réseaux dédiés *I/O* (*Storage Area Network*).

puisque les données sont inhérents à l'application. L'usage de systèmes de fichiers comme NFS peut la plus part du temps répondre aux besoins de tels programmes (les phases de calculs n'étant pas dépendantes des entrées/sorties futures, il est possible d'accepter le coût lié au chargement des données ou à la sauvegarde définitive).

Contrôle, les accès *I/O* sont distribués tout au long de l'exécution. Ce genre de flux apparaît souvent, comme son nom l'indique, lors du maniement des fichiers de contrôles (vérification du bon déroulement de l'application, réajustement de valeurs, sauvegarde succincte de l'avancement de l'exécution ...). De ce fait, les performances de l'application sont dépendantes de la fréquence et de la taille des points de contrôle apparaissant tout au long du programme. Un système de fichiers s'appuyant sur une caractérisation de ce genre d'entrées/sorties peut fournir de réels gains (nécessité d'offrir des performances optimales durant toute l'exécution de l'application).

Sur-dimensionnée ou *Out of core*, les accès aux périphérique de stockage secondaire sont dûs à la limitation de la taille de la mémoire principale. L'utilisation du concept de mémoire virtuelle[6] a permis de résoudre cette restriction. Néanmoins, cette approche à un coût non négligeable et se révèle être trop lentes pour les codes de calcul fluide. De nombreux problèmes scientifiques ont des quantités de données trop grandes pour être contenu dans l'espace de mémoire principale. Ainsi,l'augmentation de la mémoire vive, qui peut être parfois une solution temporaire, devient très vite économiquement non-viable.

1.1.3 Taille et mode d'accès

Dans un souci de compréhension et afin de bien visualiser les différents modes d'accès, nous allons rappeler les trois principales méthodes d'ouverture d'un fichier :

Lecture, les opérations réalisées seront uniquement de de type lecture.

Ecriture, les opérations réalisées seront uniquement de type écriture.

Lecture / Ecriture, les opérations réalisées seront de type lecture ou écriture.

Nous insisterons sur la notion de fichier temporaire[7] : tout fichier qui a été créé et détruit par un même nœud durant la même tache.

D'une manière générale, il a été montré que la quasi-totalité des fichiers sur des systèmes usuels (pc familiale, station de travail ...) ont une taille inférieur à 1 Mo. A contrario, les applications scientifiques hautement *I/O* intensive génèrent des fichiers de taille bien supérieur. Par ailleurs, les fichiers de type Lecture sont de dimension conséquente alors que les fichiers de type Ecriture sont pour la majeure partie de grandeur inférieure.

Par ailleurs, l'ensemble des accès sur une même fichier est souvent identique (très peu de ressources sont ouvertes afin de réaliser des opérations à la fois d'écritures et de lectures et il y a très rarement plus de quatre fichiers ouverts à la fois). Les opérations de lectures représentent plus de la majorité des accès réalisés dans le cadre des applications parallèles. Toutefois, plus de 90% des appels récupère moins de 100 octets et

[6]Une partie de l'espace de stockage secondaire est utilisée afin d'augmenter la taille de la mémoire primaire.

[7]L'usage de fichiers temporaires de type Lecture / Ecriture dans le cadre des problèmes de sur-dimensionnement n'entre pas dans cette catégorie.

seules quelques requêtes effectuent des accès sur une grosse quantité de données. Par conséquent, il est impératif de concentrer les efforts sur le problèmes de latences pour l'ensemble de toutes ces "petites" demandes.

D'un point de vue global, il y a plus de fichiers de type Ecriture que de type Lecture. Cela s'explique par la forte utilisation de fichiers temporaires sur chaque nœud (à défaut de centraliser les données au sein d'un même fichier). En effet, cette technique permet d'augmenter les performances *I/O* en utilisant les périphériques de stockages locales et en évitant les problèmes d'accès concurrents. Cependant, la programmation parallèle a imposé un plus haut degré de partage au sein d'une même ressource et il est parfois impératif de centraliser l'information en un point unique. Les problématiques d'accès concurrents souvent évitées au sein des systèmes de fichiers «utilisateurs»[8] doivent être prises en comptes dans le cadre d'applications *HPC I/O* intensive. Ainsi, un nouveau mode d'accès dit «partagé» a été identifié, il permet à un ensemble de processus d'exploiter le même descripteur de fichier [9]. Après avoir effectué de manière commune l'ouverture de la ressource, toute opération réalisée par la suite aura un impact sur l'ensemble des processus partageant le fichier.

1.1.4 Accès séquentiel et consécutif

Une requête est dite "séquentielle" si elle débute à un offset supérieur au dernier appel. Une requête consécutive est un accès qui débute exactement à l'offset où le dernier appel s'est fini. En général, les opérations au sein des fichiers de type "Ecriture" sont toujours consécutives, ceci est du au fait que ce genre de fichier est utilisé sur un unique nœud. Ainsi, il n'y pas de raison spécifique pour que l'écriture s'exécute de manière non consécutive.

Les accès recouvrants interviennent quand des données successives sont accédées par plusieurs entités du *cluster* . Du point de vue d'une unique machine , une certaine quantité d'octets doit être omise entre deux appels (appel séquentiel) alors qu'au niveau du groupe, nous pouvons observer des accès consécutifs (ces accès collectifs sont qualifiés de recouvrant). Les fichiers de type Lecture sont souvent séquentiels. L'emploi de techniques basées sur les accès recouvrants permet ceci (cf. paragraphe 1.2.1). Les fichiers de type Lecture / Ecriture ne sont logiquement pas séquentiels.

1.1.5 Régularité

Le projet CHARISMA[10][NKP+96], débuté en 1993 qui a pour but comme son nom l'indique la caractérisation des entrées/sorties, a montré qu'il existe un régularité au sein des requêtes *I/O* :

Régularité des intervalles entre chaque requêtes, la taille de l'intervalle entre chaque requête semble être régulière. Il n'y a que sur très peu de fichiers que le modèle d'accès est plus complexe (la taille de l'intervalle varie en cours de l'exécution pour un même nœud).

[8]Nous distinguons parmi l'ensemble des travaux réalisé par la communauté scientifique dans les systèmes de partages de données deux principaux groupes : les solutions à destination des «utilisateurs» (partage entre utilisateur, accès concurrents peu nombreux) et celles à destinations des applications parallèles (partage à grain fin avec une forte probabilité d'accès concurrents).

[9]"Clé" permettant d'accéder à la ressource.

[10]*CHARacterize I/O in Scientific Multiprocessor Applications.*

Remarque : Les demandes consécutives peuvent être considérées régulières avec un intervalle de zéro octet.

Régularité dans la taille de données demandée à chaque requête, la majorité des accès au sein d'un fichier se réalise sur la même quantité de données.

En combinant ces deux phénomènes, un grand nombre d'applications utilise un modèle[11] d'accès aux données clairement structuré. L'accès par plusieurs processus aux colonnes d'une matrice centralisée au sein d'un même fichier en est un très bon exemple.

Un *strided segment* est un groupe de requêtes constituant un modèle d'accès régulier. Ainsi, il est possible de définir plusieurs niveaux de régularité :

simple, tous les accès s'effectuent sur une quantité de données identique et chaque accès est séparé par le même espace (le modèle est constitué d'un unique *strided segment*).

double,triple ..., le modèle d'accès[12] est composé de plusieurs *strided segments* séparés par un même espace.

La répartition d'une matrice tri-dimensionnelle sur P processeurs peut tout à fait illustrer ces propos. En effet, si la distribution des colonnes est équitable sur l'ensemble des nœuds, le modèle d'accès est de type *simple*. Dans le cas inverse, le modèle est plus complexe et se compose de plusieurs *strided segments*, il est *double*.

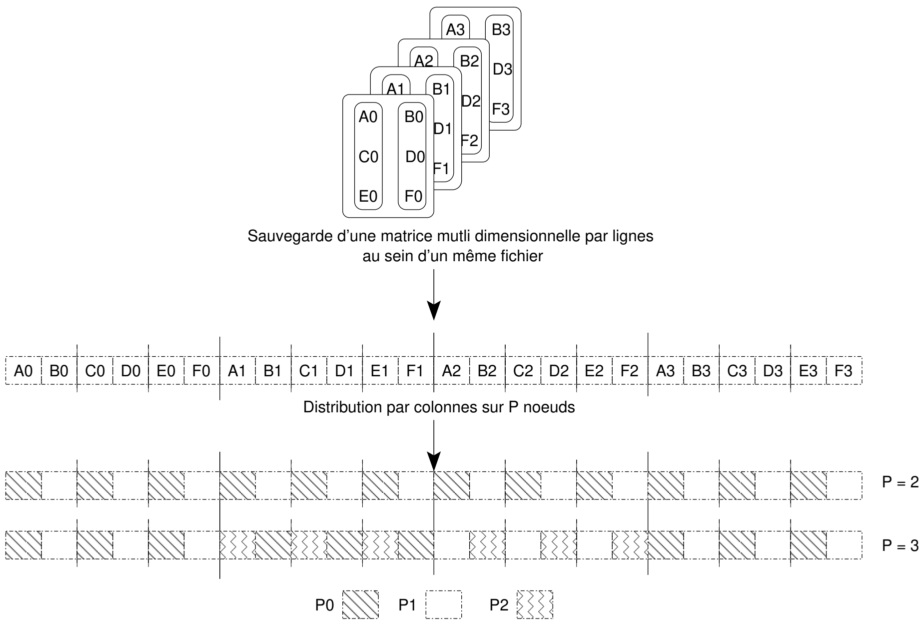

FIG. 1.1 – Distribution par colonnes d'une matrice sur un ensemble P de processeurs

Dans la figure 1.1, pour P = 2, le modèle d'accès est constitué d'un unique segment qui se décompose, après un déplacement initial[13] de la même manière pour P0 et P1 :

[11] Communément appel *strided access pattern*.

[12] *Nested pattern*.

[13] Nous remarquerons que les masques sont la plus part du temps équivalent pour l'ensemble des nœuds, seul diffère l'offset de départ sur lequel est initialisé le pointeur de fichier. Sur la figure 1.1, P0 et P1 utilisent le même masque ; toutefois le modèle pour P0 débute à l'offset 0 alors que celui de P1 est décalé d'un bloc.

1. Lecture d'une valeur,
2. Déplacement sur la prochaine valeur de la colonne (ici déplacement d'un bloc).
3. Les opérations 1 et 2 sont répétées jusqu'à la fin du fichier.

Dans le second cas où le nombre de machines est de 3, la distribution est non équitable. Même si le masque d'accès est toujours identique pour l'ensemble des processeurs, il est plus complexe et se décrit ainsi :

1. Récupération d'un segment simple
   (lecture de 2 colonnes ⇒ *simple strided pattern*),
2. Déplacement d'une colonne,
3. Les opérations 1 et 2 sont répétées jusqu'à la fin du fichier.

1.1.6 Placement réel des données

Nous avons pour l'instant observer la problématique du partage de données d'un point de vue de l'application uniquement. Il est impératif de considérer, lors du processus de partage de données, deux niveaux (cf .figure 1.2) :

Une perspective global ou logique, l'échange et le partage de données au sein d'une application (entre de multiples processus au sein d'un *cluster*). C'est le point de vue utilisé durant les paragraphes précédents.

Une vue réel/physique des données, l'échange et le partage de données au niveau des zones de stockages.

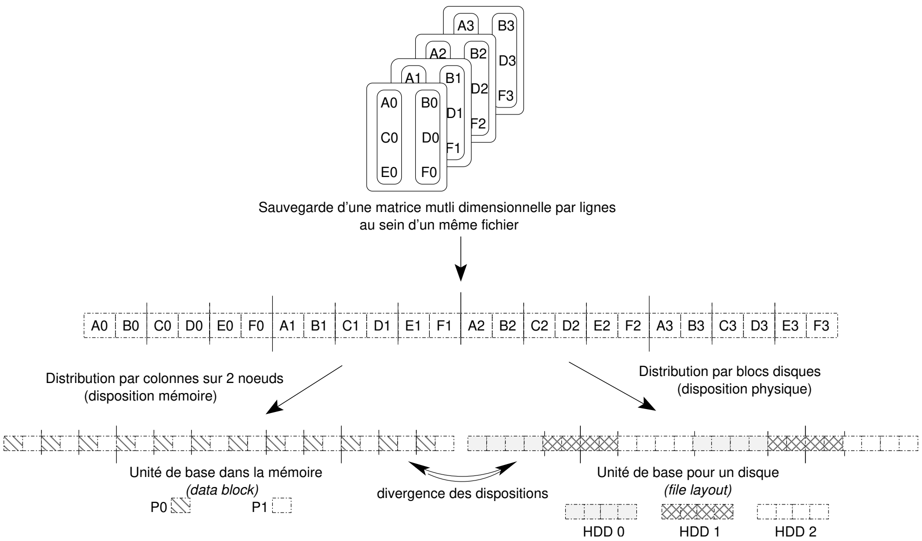

FIG. 1.2 – Différence entre le placement réel des données (contiguïté mémoire vs contiguïté physique / unité de base mémoire vs unité bloc disque).

En effet, la plupart des algorithmes que nous allons présenter par la suite (cf. paragraphe 1.2), s'appuient très souvent sur une vue global des données et ne tiennent pas compte des contraintes imposées par la structure physique sous-jacente. Ainsi, alors qu'une opération de lecture consiste en une simple copie depuis le disque vers un espace mémoire (ou de l'espace mémoire vers le disque dans le cas d'une écriture) du point de vue de l'application, il s'agit pour le système de fichier de partager des unités de données (appelées blocs disques[14]) tout en maintenant une cohérence préalablement définit.

Si le partage de ressource en lecture seule ne présente pas de concrètes difficultés[15], les opérations d'écritures peuvent entraîner de coûteuses resynchronisations lors de la prise en compte des accès au niveau du matériel dans le but de maintenir une cohérence forte[16]. En effet, la taille d'un bloc disque (ou bloc système) peut différer de celle du bloc mémoire. Ainsi, il est possible que deux processus distincts accèdent sans le savoir à des données provenant du même bloc disque (au sein de la figure 1.2, P0 et P1 travaille par exemple tous deux sur un bloc provenant du disque 0, du disque 1, ...). La gestion de ce problème, connu sous le nom de «faux-partage», entraîne un sur-coût non négligeable. Un très grande nombre d'application a été développé en tenant compte de la granularité d'un système de fichiers donné. Cette méthode relativement simple permet de contourner les problèmes énoncés ci-dessus. Toutefois, elle rend l'application non portable et ne permet pas de tirer parti par la suite des éventuels gains pouvant être apporter par les diverses couches.

Un niveau intermédiaire aux deux couches présentées précédemment, peut être inclus. Cette vue correspond à la représentation des données «cachées». Le concept de cache est une notion bien présente au sein des systèmes *I/O* . Plusieurs systèmes mettent en œuvre de tels services depuis déjà plusieurs années ; ceux-ci permettent de diminuer dans certains cas les temps de latences au niveau des interfaces *I/O* . Parallèlement aux gains susceptibles d'être perçus, l'utilisation de module de cache peut fournir une gestion plus fine de la cohérence. Toutefois dans ce dernier cas, il sera important de trouver un compromis cohérence/performance. Ainsi, les modules de gestion de caches se complexifient de plus en plus ; [VSK+03] en est un exemple où plusieurs niveaux de représentations coexistent (cache local puis global, centralisé vs réparti).

Le maintien de la cohérence est un problème majeur au sein des systèmes de fichiers ; qu'il soit ou non à destination d'applications parallèles. Ce phénomène est d'autant plus complexe et impératif qu'il est rare qu'une même et seul application s'exécute au sein d'un *cluster* à un instant donnée. Au contraire, les gros centres de calculs ont très souvent à gérer un nombre conséquent d'applications scientifiques (plus ou moins I/O intensive) ; fournir une solution capable de proposer une cohérence forte à un coût acceptable est un réel challenge de nos jours.

1.2 Agrégation des accès

Les systèmes actuels offrent des performances optimales lors d'accès conséquents : opérations de lecture (ou d'écriture) réalisées sur une grand quantité de données en une unique accès. Or, nous avons indiqué lors de la précédente section que les modèles d'entrées/sorties au sein des programmes parallèles *HPC* consiste la plus part du temps en un grand nombre de requêtes sur des petites quantités de données non contiguës.

[14]Le terme de bloc équivaut à la granularité du système de fichier. C'est la plus petite portion que le système peut remettre au disque lors d'une requête.

[15]Une copie des données est envoyée uniquement vers les zones mémoires, aucune opération n'intervient par la suite aux niveaux des disques.

[16]Certains systèmes ne proposent pas ce genre de cohérence et ne permettent, par conséquent, pas ce genre de partage.

Ainsi dans un souci d'optimisation, il est impératif de maximiser la taille des données sur laquelle les requêtes s'exécutent. S'appuyant sur des masques d'accès permettant souvent de définir ces types de demandes, plusieurs méthodes [TGL02, TGL99a, NL97, LR96, MWLY03, **?**]ont été proposées dans le but de réduire les phénomènes d'étranglement. en effet, ces derniers sont souvent amplifiés par l'envoi intempestif de petites opérations. C'est ces techniques que nous allons présenter au sein de ce paragraphe ; tout d'abord au sein d'un même nœud (cf. paragraphe 1.2.1) puis de manière collectif sur un ensemble de processus répartit sur un *cluster* (cf. paragraphe 1.2.3).

1.2.1 *Data Sieving*

Cette première méthode permet d'optimiser les accès en provenance d'un unique nœud et à destination d'un même fichier. Prenons l'exemple suivant : un processus souhaite lire cinq blocs de données non contigus ; l'utilisation de l'A.P.I. UNIX standard aurait pour conséquence l'exécution de cinq appels totalement indépendants. La solution proposée dans le concept du *Data Sieving* consiste à accéder à une portion contiguë de données recouvrant les cinq blocs (cet espace étant limité par l'offset inférieur et supérieur de l'ensemble des cinq requêtes). Les informations ainsi récupérées sont copiées dans un tampon temporaire puis les données réellement utiles sont recopiées au sein de l'espace mémoire du programme appelant. La partie droite de la figure 1.3 illustre ces propos.

Cette solution est limitée par deux principaux facteurs :

La taille du tampon de recopie temporaire , la taille du fragment nécessaire pour recouvrir l'ensemble des blocs distants peut s'avérer beaucoup plus grande que l'espace mémoire disponible. Dans ce cas, le nombre d'appels sera alors proportionnel à la taille du tampon. Cette limitation est d'ailleurs amplifié par le second paramètre.

Le degré de non continuité (cf. paragraphe 1.2.4), il définit l'espace entre deux blocs distants. Ainsi, si ce facteur est trop élevé, la mise en œuvre de la méthode peut s'avérer nettement plus cher que le coût cumulé de l'ensemble des accès requis pour récupérer les données utiles par de simples appels POSIX. [Raj02] [**?**] ont montré que le calcul de ce rapport à chaque appel permet de choisir de manière optimale la stratégie à appliquer et sans pour autant ajouter un coût discutable.

Enfin, il est intéressant de noter que si cette solution peut également être employée lors des opérations d'écritures, elle nécessite la mise en place d'un type d'accès spécifique : *read-write modified*. Ce genre de procédure permet de modifier uniquement les portions de données souhaitées (les informations comprises au sein du *strided* ne sont pas écrasées). Par ailleurs afin de garder l'intégrité des données, il est impératif de verrouiller la région du fichier qui est accédée. Par conséquent et afin de réduire les phénomènes de contentions dues à la prise de *lock*, le tampon temporaire utilisé pour les écritures est de taille beaucoup plus faible que celui mise en œuvre lors des opérations de lecture.

Remarque : L'usage de ce procédé peut être mise à défaut. En effet, l'usage d'artifices de caches ou de lecture avancée peuvent fournir une optimisation comparable au *Data Sieving* : la première requête va être exécute ; si l'espace séparant la requête suivante est assez faible, il y a de grande chance pour que les données soient déjà présentes au sein du cache lors du prochain appel. Toutefois, les coûts engendrés par de multiples demandes indépendantes sont généralement plus onéreux qu'un unique accès réalisé sur une plus grande quantité d'informations. Le cumule des deux techniques par une prise en compte du modèle d'accès (simple, double ...) au sein du gestionnaire de cache pourrait être bénéfique.

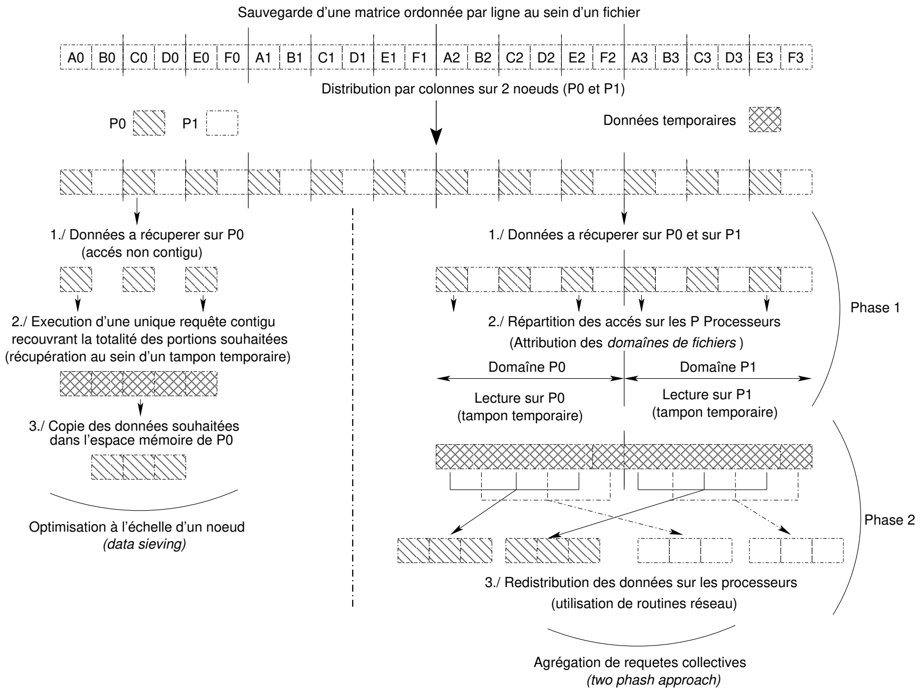

FIG. 1.3 – Techniques d'optimisation dans le cadre d'une distribution par colonnes d'une matrice sur un ensemble 2 de processeurs.

1.2.2 Stream Based I/O

A la différence de la technique précédente qui suggère une agglomération des requêtes dans le but de maximiser la bande passante disque, cette approche se concentre sur les contraintes réseaux et les problèmes de contentions. Développée dans le cadre du projet *PVFS* ([LR96], cf. paragraphe 2.2.4), le concept consiste à agréger plusieurs requêtes au sein d'une même trame réseau afin de l'envoyer par la suite au serveur de données. Le nombre de messages transitant ainsi sur le *cluster* est considérablement diminué.

La construction de chaque trame "de requêtes" prend en compte le protocole d'échange employé (*TCP/IP*,*Myrinet*, ...) et la taille optimale des segments correspondants. Ainsi, cette méthode permet de réduire considérablement les messages de contrôle nécessaire à chaque requête *I/O* mais également de diminuer le transit réseau. En effet en comparant à la méthode *Data Sieving*, seules les données «utiles» (trame communément intitulée *Stream I/O*) circulent entre les diverses entités (les données comprises entre chaque portion souhaité ne sont pas récupérées ou transmises lors des écritures). L'opération d'empaquetage des demandes *I/O* (coté client) et la phase de restructuration des requêtes (coté serveur) représentent les seuls «coûts» additionnels de cette méthode.

Néanmoins, cette solution ne prends pas en compte les problématiques liées au placement physiques des données (il est possible d'effectuer plusieurs requêtes sur des mêmes données). Par ailleurs, elle été proposée en s'appuyant sur une architecture où la latence réseau est relativement élevée par rapport aux performances *I/O* des unités de stockages ce qui à ce jour se révèle souvent inexacte. Un compromis intéressant entre les concepts de *Stream Based I/O* et *Data Sieving* permettrait de bénéficier totalement des capacités proposées par l'architecture matérielle sous-jacente.

1.2.3 *Collective I/O*

Dans de nombreuses applications parallèles, chaque nœud peut accéder à plusieurs parties non contiguës d'un même fichier. A la différence du cas précédent où l'optimisation était réalisée indépendamment des autres processus, nous nous plaçons ici dans un contexte de requêtes en provenance de plusieurs machines. L'intérêt est de tenir comptes des accès parallèles recouvrants lors de demandes collectives. Une agrégation optimale de ce type de requêtes peut apporter un réel gain ; ce concept est communément appelé *Collective I/O*.

L'agglomération des opérations peut être implémenter à plusieurs endroits, chaque solution possédant évidemment ses avantages et ses inconvénients :

Au niveau des clients, deux techniques ont été proposées : *Two Phase I/O*[dBC93] et plus récemment *Collective Buffering*[NL97] :

La première initiative est celle la plus régulièrement mise en œuvre. Elle est notamment implanter au sein de *ROMIO*, la couche *MPI I/O* de *mpich* (le processus est décrit en détail au sein du chapitre**??**). Comme son nom l'indique, le modèle se divise en deux principales parties :

1. Les processus s'informent mutuellement des futurs opérations d'entrée/sortie puis les exécutent d'une manière à maximiser la taille des requêtes
2. Les données sont redistribuées entre les divers processus.

Cette solution brièvement décrite dans la partie gauche de la figure 1.3 s'appuie sur une architecture possédant un réseau rapide (les *I/O* disques s'avèrent beaucoup plus pénalisant que les *I/O* réseaux). Pour les opérations d'écriture, les deux étapes sont simplement inversés.

La seconde technique diffère peu du concept énoncé précédemment. Ainsi, les procédures dites de «*Collective Buffering*» ont pour principal but l'optimisation de l'étape de redistribution. La totalité des nœuds est concernée dans l'approche généralisée de l'algorithme *Two Phase* (envoi + réception et copie dans un tampon temporaire sur chaque unité) ; ici la phase de permutation des données est améliorée :

1. En tenant compte de la distribution des données en mémoire (distribution par bloc, cyclique ...) et de la taille des tampons temporaires (le but étant de minimiser l'espace mémoire requis),
2. En réduisant les flux réseaux, pour cela des primitives[17] permettant d'agréger au sein d'un même message plusieurs informations (à chaque itération, un seul message contenant la totalité des blocs de données à permuter est émis vers le nœud cible au lieu d'envoyer une message pour chaque information).

[17] *Scatter/Gather*.

D'une manière générale, il n'y a plus de réelle distinction entre ces deux méthodes. L'algorithme *Two Phase* tient compte des diverses mises au point proposées par l'ensemble de la communauté scientifique et permettant de maximiser le ratio coût/performance.

Au niveau des disques, ce procédé connu sous le nom de *disk-directed I/O*,[Kot94], repose sur le modèle décrit précédemment. Il suggère la mise en place d'un module permettant l'agrégation des appels selon des critères spécifiés (latence *I/O* , placement mémoire vs positionnement physique des données, débits réseaux . . .). L'algorithme se résout alors en une unique phase d'émission/réception :

1. Des demandes «simples»[18] sont transmises sur l'ensemble des disques où elles sont agrégées afin de maximiser les performances puis enfin exécutées.
2. Les données lues sont alors directement retransmises sur le nœud souhaité.

La phase de redistribution inter-nœud ayant disparue, les performances s'en révèlent meilleures. Toutefois, cette approche est nettement moins portable que les méthodes soumises auparavant. Une étude [IRU01] s'est inspiré de critères d'optimisation (localité des données, consommation restreinte des flux réseaux) afin d'améliorer l'implémentation de *ROMIO* pour le système de fichiers *PVFS*.

Au niveau du/des serveur(s), le modèle *server-directed I/O* a été développé dans le cadre du projet *PANDA*[19][SCJ$^+$95] ; il correspond à une abstraction du concept *Disk Directed I/O*. En effet, ci ce dernier apporte de réels gains lors des accès, il reste toutefois très dépendant de l'architecture sous-jacente. L'intérêt de remonter ce module à un niveau plus élevé dans la pile système améliore la portabilité du modèle et ce s'en pour autant rajouter un coût considérable. Cette dernière technique est toujours d'actualité.

Remarque : [AUB$^+$96] montre qu'il est possible d'obtenir les performances requises au sein des applications *HPC* sans passer par des approches de type *Collective I/O*. Tout d'abord, il montre que l'utilisation de techniques collective augmente les phénomènes de latences en imposant de nombreux points de synchronisation pour débuter les différentes phases (temps pendant lequel l'unité de stockage n'est pas exploitée). S'appuyant sur quatre applications scientifiques correctement remaniées pour maximiser l'utilisation des disques («larges» requêtes *I/O* , exploitation d'opération de *Data Sieving* via la librairie *Jovian 2*, cf paragraphe 2.3.4), il prouve qu'un gain comparable peut être apporter. Toutefois, il est nécessaire d'intervenir au sein du code afin d'apporter les modifications permettant ces optimisations ; ce qui complexifie, là encore, la tâche du développeur. Il paraît donc important de prendre en compte les remarques citées en vers les opérations collectives et continuer à travailler sur des interfaces de plus haut niveaux permettant de rendre plus aisé la programmation d'application parallèle.

1.2.4 Degré de continuité

Comme nous l'avons indiqué, le temps nécessaire pour accéder aux informations varie selon les méthodes mise en place. L'utilisation de technique d'agrégation améliore nettement les performances. Toutefois dans certains cas critiques, l'approche habituelle UNIX s'avère moins coûteuse. Nous allons illustrer ces propos en présentant

[18]Les données ne sont pas directement transmises lors des opérations de lecture ou d'écritures. Le module *disk-directed I/O* aura à sa charge la récupération des informations.

[19]*PANDA* est une librairie qui a pour but l'optimisation des *I/O* au sein des tableaux à plusieurs dimensions sur des architectures aussi bien séquentielles que parallèles, cf paragraphe 2.3.2.

la quantification réalisée au sein de [Raj02]. Dans un premier temps, nous mesurerons une lecture de n blocs non contigus à partir d'un unique processus puis nous réaliserons la même opération de manière collective.

Quantification au sein d'un nœud :

1. Approche UNIX standard : n déplacement(s) et n lecture(s)

$$T_1 = n\left(S + \frac{C}{D}\right) \quad \text{(1.1)}$$

avec S, temps de positionnement sur l'offset (*seek*),
C taille d'un bloc en Mo,
D débit en Mo/s et n nombre total de blocs.

2. *Data Sieving* : Agrégation à l'intérieur d'une machine

$$T_1' = S + \frac{C'}{D} + nM \quad \text{(1.2)}$$

avec C' taille du bloc de recouvrement en Mo
(au maximum la taille du tampon temporaire)
et $M = \frac{C}{Q}$ temps nécessaire pour la copie temporaire d'un bloc utile (*memcpy*).

Dans la plus part des cas, la relation suivante est vérifiée :

$$T_1' \ll T_1 \qquad \equiv \qquad S + \frac{C'}{D} + n\frac{C}{Q} \ll n\left(S + \frac{C}{D}\right) \tag{1.3}$$

Toutefois, si les données «utiles» comprises au sein du bloc recouvrant se révèlent trop éloignées, il est possible que le coût de la lecture de la totalité du bloc recouvrant soit plus onéreuse que le coût généré par l'approche 1. Nous avons alors :

$$\frac{C'}{D} \gg n\left(S + \frac{C}{D}\right) \qquad \Rightarrow \qquad T_1' \gg T_1 \tag{1.4}$$

Il en est de même si la taille des portions non contiguës est supérieure ou égale à la taille du tampon temporaire de recopie (il n'est alors pas utile de copier les données au sein du tampon temporaire).

$$T_1' = n\left(S + \frac{C'}{D} + \frac{C}{Q}\right) \qquad avec \qquad C' \geq C \qquad \Rightarrow \qquad T_1' \gg T_1 \tag{1.5}$$

Ces situations où les méthodes d'agrégations des accès ne sont pas performantes se retrouvent également lors de l'approche collective. quantifions à son tour ce genre d'opération en nous appuyant sur la méthode *Two Phase*.

Quantification au sein d'un groupe de processus :

1. Approche UNIX standard : p processeur(s) effectue(nt) n déplacement(s) et n lecture(s)

$$T_n = p * n\left(S + \frac{C}{D}\right) \quad \text{(1.6)}$$

avec S, temps de positionnement sur l'offset(*seek*),
C taille d'un bloc en Mo, D débit en Mo/s,
p nombre de processeurs et n nombre total de blocs.

2. *Collective I/O*, *Two Phase* : Agrégation à l'échelle d'un groupe de nœuds

$$T_n' = pn\frac{R}{B} + pT_1' + pn\frac{L}{B} \quad \text{(1.7)}$$

avec B, débit fournit par le médium réseau,
R, taille d'un message d'échange *I/O*
et L, taille d'un message de redistribution.

En posant l'hypothèse que le médium réseau est parfait (l'opération de multicast se réalise en un temps constant) :

$$T'_n = pn\frac{R}{B} + pT'_1 + pn\frac{L}{B} \equiv T'_n = n\frac{R}{B} + pT'_1 + n\frac{L}{B} \tag{1.8}$$

De même, les p accès réalisés en T'_1 ne sont plus recouvrants et peuvent par conséquent être parallélisés. Ainsi le temps total pour réaliser l'opération collective s'exprime par la formule suivante :

$$T'_n = n\frac{R}{B} + T'_1 + n\frac{L}{B} \tag{1.9}$$

En s'appuyant sur une latence très faible fourni par les réseaux actuels (myrinet...), le coût engendré par la réalisation d'une demande collective est alors corrélé aux latences *I/O* et par conséquent à l'espace qui sépare l'ensemble chaque fragment de données (remarque que nous avons notamment précise lors du *Data Sieving*). Il est donc impératif de mettre en place un coefficient permettant de quantifier à la volée les méthodes et choisir la stratégie optimale.

Mise en place d'un coefficient :

Le coefficient de continuité ou «degré de continuité» permet de déterminer la technique appropriée afin d'optimiser les accès. Ce paramètre est calculé en s'appuyant sur le temps séquentiel, sur le temps parallèle pour les opérations non collective et enfin sur le temps en utilisant les A.P.I. collectives.

$$\beta = 1 - \frac{min(T'_1, T'_n)}{T_S} \tag{1.10}$$

De cette manière :

si $\beta < 0$ le style UNIX est recommandé,
$\beta \to 1$ l'agrégation est alors recommandé.

A ce jour, cette estimation n'est pas inclue au sein de l'implémentation ROMIO. Un test élémentaire, présenté dans le chapitre **??**, est uniquement mis en œuvre lors de l'approche *Two-Phase*.

1.3 Entrées / sorties asynchrones

Les techniques d'agrégation présentées précédemment permettent d'exploiter efficacement les débits proposés par les unités de stockages actuelles. Les latences liées aux opérations d'entrée/sortie sont minimisées mais toutefois toujours présentes. Une approche complémentaire à l'agglomération subtile des demandes *I/O* consiste en l'usage de techniques asynchrones permettant soit de prévoir, à l'avance, d'éventuelles demandes (optimisation pouvant subvenir lors de la phase de compilation ou lors de l'exécution via des techniques de cache et de «lecture avancée») soit par l'exploitation de diverses fonctionnalités (logiciel ou matériel) capable de réaliser des opérations en arrière plan. Nous verrons qu'un grand nombre de solutions (*Galley*, *PVFS*^{2}, *MTIO*, MercutIO ou encore *MPI I/O* cf. chapitre 2) mettent en œuvre des méthodes de ce type afin de rendre plus «transparent» le coût des opérations *I/O* en maximisant le recouvrement *I/O* / Calcul.

Les fonctionnalités offrant l'asynchronisme sont intégrées de manière croissante au sein des disques de stockages. Cependant peu de systèmes exploitent d'ores-et-déjà cette capacité et les opérations asynchrones sont implémentées en s'appuyant sur des processus s'exécutant en arrière plan[20]. Cependant [DT99, DT01] révèlent l'importance une fois encore d'une étude rigoureuse et d'une exploitation optimale des *threads*. En effet dans le cadre des opérations collectives basées sur la méthode *Two-Phase*, l'approche naturel consistant à créer un *thread I/O* afin de réaliser l'opération en arrière plan (pendant que le processus principal continue avec d'autres calculs) s'avère souvent plus laborieuse qu'exécuter la totalité de l'opération *I/O* au premier plan. En effet, l'algorithme *Two-Phase* comprend un certain nombre d'étapes pouvant être recouverte par du calcul, l'opération *I/O* en elle-même mais également lors de la phase de redistribution des données (après l'envoi et jusqu'à la réception des messages de permutations des données). Ainsi, une fois que les deux *threads* sont créés, ils rentrent en «compétition» pour le contrôle du *CPU* : le coût engendré par les changements de contexte[21] et la forte utilisation de l'ordonnanceur n'est pas avantageux lorsque le recouvrement est réalisé sur une période trop courte.

La méthode optimale repose sur une répartition des deux étapes ; la phase de redistribution est réalisé au sein du processus principale alors qu'un *thread* effectuera uniquement l'opération *I/O* effective en arrière plan. Il est toutefois important de noter l'importance de la taille du tampon temporaire permettant de réaliser le code de redistribution. En effet, si l'espace de recopie n'est pas assez grand, plusieurs itérations comprenant les deux étapes vont être effectuées. Dans ce cas, il s'avère plus performant d'accomplir la totalité de l'opération collective au sein du processus principale.

Toutefois, l'usage de routines asynchrones impose un nouveau fardeau aux développeurs d'applications parallèles devant déjà prendre en compte les problèmes de vue logiques et placement physiques. Pour toute opération asynchrone, il est impératif de prévoir un code de recouvrement ce qui peut complexifier nettement la programmation.

1.4 Caches et techniques de pré-chargement

L'utilisation d'espace de stockage locale est un moyen supplémentaire pour limiter les phénomènes de saturations au sein des applications *I/O* intensive. Depuis fort longtemps éprouvé, le concept fondamentale consiste à sauvegarder («cacher») les données au sein de la *RAM* ou le cas échéant sur la mémoire secondaire afin de diminuer les latences *I/O* tout en limitant les interactions avec les unités de sauvegardes distantes (serveur de données, baies de disques, ...). L'exploitation de tampons temporaires[22] couplée à diverses méthodes est employée afin de répartir aux mieux les charges sur l'ensemble du *cluster* et éviter ainsi tous phénomènes d'étranglement. De nombreux modèles ont intégrés une politique de cache au sein de leur architecture (une brève synthèse apparaît dans [Leb02]), l'omniprésent *NFS* propose un cache client basé sur

[20] L'aboutissement des systèmes d'exploitation multi-tâches incluant la gestion des *threads* a largement contribué à ce phénomène.

[21] Il est important de différencier deux types de *thread* :
Utilisateur, peu coûteux mais ne permettant de recouvrir *I/O* / calcul,
Noyau, permettant réellement du multi-tâches au sein d'un processus mais coûteux.

[22] *Buffer Cache*.

des délais de garde afin d'assurer une pseudo-cohérence des données entre les différentes images (le serveur dit «*stateless*» est totalement indépendant de cette partie du système implémentée sur chaque client). Les approches de type *AFS* intègrent également un niveau de cache sur leurs clients ; toutefois la cohérence est maintenue par l'intermédiaire du serveur de données (serveur «*statefull*»). Cette méthode qui offre un niveau de sémantique plus fort engendre, en plus d'une surcharge sur le serveur, un traffic réseau supplémentaire (échange de messages d'invalidation entre les entités qui composent le *cluster* , *callback*). Les architectures comme *CODA* ou *Intermezzo* ont continué à développer ce concept en intégrant au sein de leurs modèles, des gestionnaires de caches persistants offrant ainsi un très haut coefficient de disponibilité (possibilité de travailler en mode dit «déconnecté»). Toutefois, les méthodes utilisées afin de resynchroniser les images ne peuvent être appliquées sans apporter quelques modifications pour pouvoir être exploitées au sein des applications *HPC* (le système de fichier *Lustre* a intégré tout récemment ces concepts, cf. paragraphe 2.2.5).

La thématique des caches fonde à elle seule un axe de recherche complet et complexe. J'ai choisi de présenter d'une manière très synthétique les techniques majeures pouvant être incluses au sein des gestionnaires de caches (cf.tableau 1.1). Chaque approche décrite au sein du tableau insiste sur des critères spécifiques et parfois très distincts ; l'intéret n'est pas de regrouper les méthodes afin de les comparer mais d'informer le lecteur sur les approches fondamentales et la terminologie qui s'y rattachent.

Méthode	**Description**
Least Recently Used	Les *buffer caches* sont modélisés la plupart du temps par des listes chainées de taille limitées. Lorsque la liste est saturée, tout ajout de bloc nécessite l'éviction d'un autre ; la politique *LRU* retire la portion de données la moins récemment utilisée. Diverses améliorations ont été proposées par la suite afin de déterminer de manière plus optimale le bloc à supprimer (*LRU-Interleaved*, *LRU-SP*, ...).
Read ahead	Opération de lecture par anticipation communément apellée *Prefetch*, la technique consiste a déclencher une opération préalablement afin de rendre transparent tout latence *I/O* (la requête d'entrée/sortie est réalisée par anticipation de manière à ce que les données soient présentes au sein du cache lorsque l'application en aura réellement l'utilité). Un tri entre les requêtes de *prefetch* peut-être éventuellement intégré à cette politique (*disk scheduling*). Le gestionnaire agrégé dans ce cas un ensemble de requêtes «asynchrones» de manière à optimiser la bande disque.
Write delayed	Opération d'écriture «retardée». Le *buffer cache* est utilisé afin d'éviter tout interaction stérile avec une unité distante (cas des fichiers temporaires qui n'ont pas à être archivés, modifications intempestives d'un fichier par un même écrivain, ...).Il est intéressant de noter ici la possibilité de coupler cette approche à l'utilisation de tâches d'arrière plan (opérations asynchones) ; les opérations d'écritures peuvent alors être totalement transparente.

Méthode	**Description**
(No) Write through	Toute opération d'écriture est propagée directement sur l'unité de stockage correspondante. L'interêt est d'assurer une cohérence stricte. Au contraire, la technique de cache persistant (*No write through*, *AFS*, *CODA*) a pour objectif de proposer un coefficient de disponibilité maximum tout en limitant au stricte minimum les échanges avec les unités de sauvegardes distantes. La cohérence entre les «images» est assuré via des opérations de resynchronisations («vecteurs d'historiques»).
Client to Client transfers	Exploitation des caches clients distants Les serveurs de données redirigent les requêtes vers le client qui possèdent déjà une image des données (la charge est ainsi mieux répartit sur la totalité du *cluster*).[CFKL96] descendant du projet *SPRITE* s'appuie notamment sur ce type d'approche.
Cache local à deux niveaux	Le gestionnaire se scinde en deux sous-modules. Le premier, intégré au niveau du noyau, répartit l'espace disponible entre les différents processus en appliquant une méthode de type *LRU*. Le second sous-système, inclut généralement au sein des librairies d'optimisations de plus haut niveau, offre une gestion plus fine de la partie d'espace allouée (*Prefetch*, ...).
Cache local vs Cache global	Une nouvelle entité est intégrée au système de partage de données afin d'assurer les services de cache. Plusieurs modèles ont été suggérés ; le serveur de données transmet par exemple une copie à un nœud de l'architecture qui centralisera l'ensemble des requêtes en provenance des clients. Une seconde approche consiste à laisser les opérations de maintenance du cache à la charge des clients ; l'unité centralisé ayant uniquement à sa charge, les opérations de localisations des différentes images (le serveur de données redirige la requête au serveur de cache qui va indiquer au client ou retrouver la donnée).
Cache hiérarchique	Mise en place de plusieurs gestionnaires intervenant à plusieurs niveaux au sein de l'architecture (cache local sur les clients, global par des entités intermédiaires et enfin cache présent sur les serveurs de données et/ou les unités de stockages).

TAB. 1.1: Techniques et méthodes

Néanmoins, l'usage de caches n'est pas toujours bénéfique. En effet, s'il augmente généralement considérablement les performances, la mise en place de certaines techniques peut s'avérer défavorable. La figure 1.4 en présente un exemple (utilisant de techniques de type *Prefetch* au sein d'un cache de taille limitée). Cette illustration nous permet de montrer l'étroite corrélation entre les paramètres physiques du *buffer cache* et les performances pouvant être obtenues.

En effet, pour les paramètres suivants :

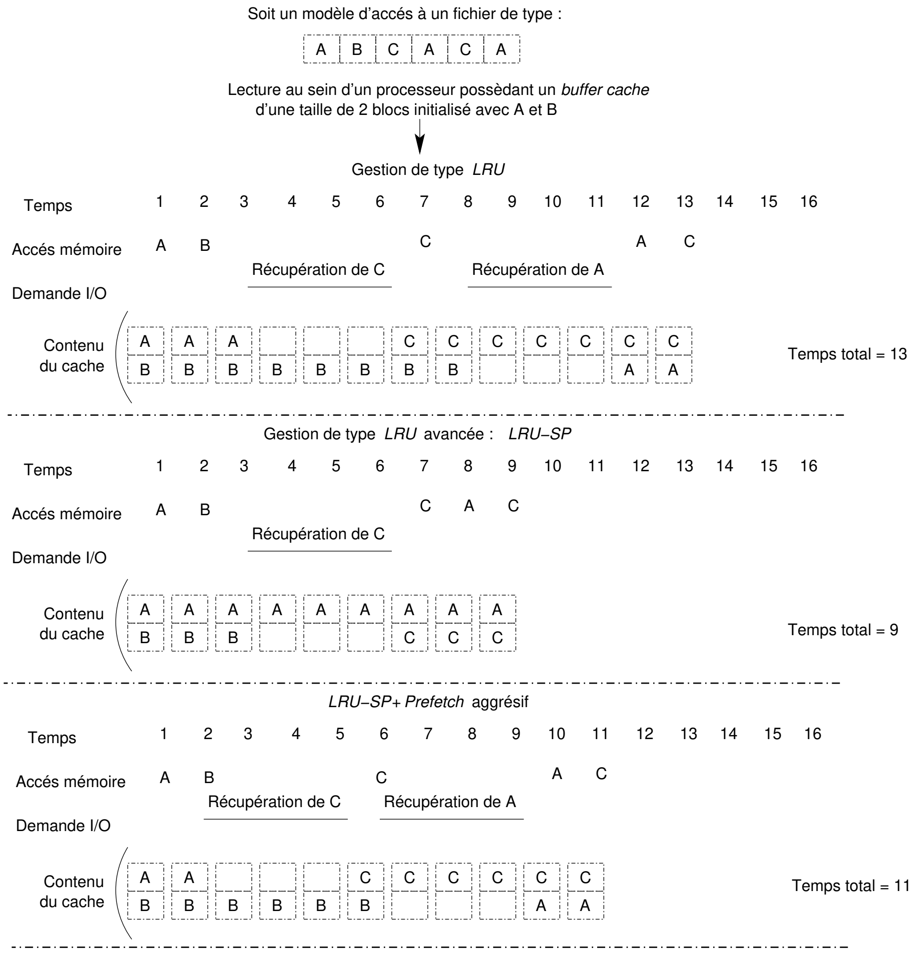

FIG. 1.4 – Cache et *Prefecth* : Performance et Contre-performance

opérations de lecture selon le «patron» : $ABCACA$ au sein d'un processus exploitant un *buffer cache* pouvant contenir deux blocs disques,
le temps nécessaire pour accéder à la mémoire est de une unité de temps,
le temps de récupération d'un bloc est de quatre unités.
Nous obtenons 13 unités pour réaliser la tâche en exploitant une méthode de type *LRU*[23]. Neuf unités sont nécessaires lorsque l'algorithme *LRU* prend en compte les futurs accès[24]. Enfin, nous pouvons observer que la mise en œuvre de méthodes de type *Prefetch* n'apporte pas toujours un gain. Dans notre cas, la taille du *buffer* impose que la donnée A soit éliminée du tampon (B étant utilisée lors du déclenchement de l'opération de récupération C, il est impossible de le retirer comme l'indiquerait l'approche *LRU-SP*). Par ailleurs, la taille du *buffer* et le moment où l'opération de récupération par anticipation peut être déclenchée sont fortemment liés (sur notre exemple, une seule

[23]La politique de cache utilisée par les systèmes de fichiers traditionnels.

[24]Dans ce cas précis, il est impératif de transmettre le modèle d'accès afin que le gestionnaire de cache puisse appliquer la stratégie optimale.

unité de temps peut être gagné).

L'apport de connaissance, comme dans le cas des opérations collectives, autorise le gestionnaire de cache à opter pour une stratégie optimale en tenant compte, d'une part, de ces paramètres physiques, et d'autres part, du modèle d'accès en provenance de l'application. Malheureusement, les routines standards UNIX ne fournissent pas ce genre de renseignements ; l'utilisation d'appels spécifiques est donc requis [25].

Parallèlement à ces critères locaux (ou internes à un *buffer cache*), il est important de remarquer que la mise en place de plusieurs tampons à de multiples niveaux peut engendrer dans le pire des cas un surcôut non négligeable. En effet, si chaque interrogation (émission de la requête + recherche au sein du tampon) émet un défaut de page, l'opération *I/O* sera finalement exécutée. Le coût total se composera alors de la latence *I/O* additionnée aux latences des caches. [VSK+03] suggère une solution permettant de définir le meilleur «chemin» en outre-passant certains niveaux le cas échéant.

Les algorithmes qui composent les gestionnaires de caches se complexifie de manière exponentielle lorsque ceux-ci sont distribués sur l'ensemble de l'architecture et qu'ils doivent prendre en considérations des demandes provenant de plusieurs processus (recouvrement, «faux-partage», intégrité entre les multiples copies, ...). Cependant, il est primordiale pour les nouveaux systèmes de proposer des politiques de caches fiables et efficaces ; un compromis entre gain, coût (et complexité) doit être clairement définit lors de la mise en place de tels modules.

1.5 Vers un système de gestion de données parallèles

Depuis quelques années, une résurgence de systèmes de partage de données est apparue. *PVFS* [CLRT00] , *GPFS* [Sch02] ou encore le tout récent Lustre[26] en sont les actuels porte-flambeaux. Chacun s'appuie sur des technologie ou des concepts qui diffèrent mais tous tentent de répondre aux critères de performances requis par les applications scientifiques hautement parallèles. [TGL99b] ont présenté les aspects qui permettraient à un système de fichiers parallèle couplé à la libraire *MPI I/O* d'offrir des performances optimales. Certains de ces considérations doivent être impérativement prises en compte afin de proposer des solutions profitables pour l'univers *HPC* . Nous les avons répertoriés ci-après :

1. Sémantique de partage, appelée aussi consistance des données ; elle doit être clairement définit et correctement implémentée afin de prévoir le comportement du système en cas d'accès concurrent en provenance de plusieurs nœuds.
   Cette cohérence doit être présente aussi bien au niveau des données qu'au niveau des méta-informations (attributs concernant les fichiers : taille, estampille de temps, ...). Proposer une sémantique POSIX[27] au sein d'un système peut être un réel atout. La sérialisation des opérations d'écritures est par exemple un réel problème si les granularités mémoire/disque diffèrent.

[25] Même si nous avons cité lors de l'introduction que certains travaux proposent diverses solutions permettant de récupérer ces informations durant la phase de compilation (graphe de flot de données). L'irrégularité de certains problèmes rend impossible la mise en œuvre de telles méthode.

[26] http ://www.lustre.org

[27] Toute opération d'écriture sur un bloc disque est immédiatement visible par l'ensemble des processus une fois celle-ci terminée.

2. Atomicité des opérations, elle doit pouvoir être ou non activée ; le but étant de ne pas pénaliser les opérations indépendantes (qui ne se recouvrent pas et donc qui n'ont pas besoin d'être atomique.)
3. Système de *byte-range locking*, ce type de service permet de verrouiller une plage d'octets au sein d'un fichier et permettre de réaliser des opérations protégés (le système GPFS, en autre, propose ce type de service).
4. Interface d'accès à une portion de données non-contiguës, implémentation de l'algorithme *Data Sieving* (cf. 1.2.1) au sein d'une couche interne au système d'informations.
5. Distribuer l'ensemble des couches qui composent le système de fichier en évitant au maximum les points de centralisation. En effet une unité centralisée peut rapidement devenir un goulet d'étranglement pour un système composé d'un grand nombre de nœuds[28].
6. Mode d'attribution des données sur l'ensemble des serveurs de stockages dynamique ; l'intérêt majeur est d'adapter la granularité de répartition des données par rapport à celle des accès de chaque application (toujours dans une optique de maximiser la parallélisation des requêtes et réduire les temps de latence).
7. Supporter les fichiers supérieurs à 2 Go (architecture 64 bits) ; les tailles de fichiers ont considérablement «explosé», il est donc fondamentale de supporter des tailles supérieurs aux 2Go proposées par la plupart des systèmes actuels.
8. Pré-allocation des zones de stockages ; les opérations de créations sur l'ensemble des méta-informations sont exécutées préalablement et n'auront pas à être réalisées lors de la création effective des nouveaux fichiers (les temps de latence sont là encore réduit).
9. Politique de caches variable ; en effet comme nous l'avons indiqué précédemment, si les techniques de caches sont mises en place dans le but d'optimiser les accès, elles peuvent s'avérer dans certains cas, plus onéreuses et inappropriées. Il est donc nécessaire que les modules de caches (cf. pararaphe 1.4) de prochaine génération prennent en compte des paramètres liés au comportement de l'application et à l'architecture sous-jacente afin de proposer des optimisations réellement efficaces.

L'ensemble des considérations précédemment décrite peut être complété par trois autres aspects toutefois plus optionnels :

10. Mise en place de pointeurs de fichiers partagés ; comme nous l'avons brièvement introduit au paragraphe 1.1.3, un fichier peut alors être accédé par de multiples clients utilisant le même descripteur (tout déplacement du pointeur est répercuté sur chaque client). Toutefois, la plus part des librairies *Parallel I/O* inclut cette fonctionnalité afin de ne pas surcharger le système de fichier. Dans le cas contraire, le système doit maintenir en cohérence une table pour chaque pointeur partagé ; cette table contenant, en plus de la position et des informations relatives au descripteur, les identifiants de chaque client accédant aux fichiers.
11. Opérations *I/O* asynchrones ; une grande majorité des librairies incorpore ce service en utilisant des processus en arrière-plan (*thread*). Toutefois, certains nouveaux matériels proposent cette fonctionnalité, il est donc intéressant pour les performances que le système de fichiers les utilisent directement.

[28]Le problème de tolérance au panne en étant lui même affecté : *Single Point Of Faillure*.

12. Gestion des opérations collectives dans le cadre de serveur centralisé ; nous avons mentionné l'importance de distribuer la totalité des entités qui composent le système au point 5./. Cependant, un grand nombre de serveurs s'appuient sur une couche centralisé[29]. Dans ce cas précis, il faut impérativement exploiter ce point centrale pour proposer une optimisation a l'attention des opérations collectives (cf. 2.3.2, la totalité des demandes *I/O* transitant par le nœud, il est alors possible de choisir la meilleure stratégie d'agrégation en ce point).

D'un point de vue globale, la prise en compte, premièrement de l'architecture matérielle sous-jacente (capacité d'analyse, débits disques et réseaux, topologie de l'architecture) et deuxio du comportement de l'application est un facteur primordiale pour la génération future des systèmes de fichiers. Parallèlement, si la mise en place d'une cohérence plus ou moins forte mais clairement définie et correctement implémentée est dores-et-déjà un problème ; proposer une cohérence fine sans altérer les performances est un réel challenge !

L'utilisation de librairie complémentaire aux systèmes de partage de données est une première approche dans le cadre de la programmation d'applications parallèles I/O intensives. Une interaction optimale entre ces deux «modules» est donc capitale. Nous allons dans le chapitre suivant présenter les travaux effectués dans ces deux univers depuis une dizaine d'années.

[29]Ces modules sont souvent dupliqués pour prévoir toutes éventuelles pannes. Néanmoins, les clients ne voient pas ces entités «miroirs» ; toutes requêtes transitent par l'unité centralisée active

Chapitre 2

Evolution des travaux et solutions actuelles

L'enthousiasme pour le développement des *CPUs* et celui des réseaux durant les deux dernières décades a largement contribué à renforcer la disparité de performance qu'il existe avec les systèmes de stockages. Ainsi, au milieu des années 90, l'éclosion de multiples systèmes massivement parallèles à dévoiler l'insuffisance des interfaces disponibles à l'époque (inapproprié d'un point de vue des débits bien sur, mais également aux niveaux des capacités et des interfaces d'utilisation). De nombreux projets ont vu le jour ; dans un premier temps, ils ont eu pour but la caractérisation du comportement des applications modernes (cf chapitre 1). Puis par la suite, plusieurs solutions et techniques d'optimisation s'appuyant sur les travaux précédents, ont été proposées ; les plus représentatives sont décrites au sein de ce chapitre. Les spécifications *MPI I/O* qui découlent de l'ensemble de ces travaux et qui s'impose peu à peu comme le modèle officiel de programmation parallèle pour les applications *I/O* intensive sont décrites au sein d'un troisième chapitre.

2.1 Généralités

Comme nous l'avons cité lors de l'introduction, plusieurs techniques ont été proposées afin de pallier au manque de performance. Parmi les approches tentant de résoudre cette problématique durant la phase d'exécution [1], nous pouvons distinguer deux principales sous-catégories : la première consiste à concevoir et mettre en œuvre de nouveaux systèmes de fichiers «parallélisés». Ces derniers exploitent certains concepts présents au sein des technologies *RAID* [CLG+94] [2] afin de réduire les phénomènes d'étranglement. A la différence, la seconde démarche s'appuie sur les systèmes déjà présents et les complète par l'intermédiaire de librairies de plus haut niveau. Chacune de ces solutions comporte des inconvénients (dépendances vis à vis des couches internes, par rapport aux matériels, ...) et des avantages (portabilité, optimisation bas niveaux, ...).

Un premier paragraphe rassemble l'ensemble des systèmes de fichiers dits "paral-

[1] Nous rappelons que certains travaux proposent des méthodes d'optimisations qui apparaissent lors de la compilation mais qu'il a été choisi volontairement de ne pas les traiter au sein de ce rapport.

[2] La méthode du *file stripping* parallélisées majoritairement employée.

lèles"[3]. Le second s'attarde, quand à lui, sur les libraires majeures qui ont été soumises à la communauté scientifique. Enfin et pour conclure, nous décrirons à la fin de ce chapitre l'*ADIO*[4] ; cet intergiciel apporte la cohésion nécessaire entre toutes les solutions précédemment présentées. Il permet d'abstraire par l'intermédiaire d'une couche centrale les multiples *APIs* de haut niveaux par rapport au systèmes de fichiers plus dépendants des architectures matérielles sous-jacentes (cf paragraphe 2.4).

2.2 Système de fichiers «parallèles»

Un des avantages principale de cette approche est une utilisation «optimale» de l'architecture matérielle sous-jacente. En effet, ces nouveaux systèmes re-développent généralement la totalité des couches qui les composent. Ainsi, il leur est possible de contrôler la totalité des interactions et d'exécuter divers types d'optimisation (agrégation, cache, ...) tout au long du processus d'entrée/sortie.

Les solutions mise en œuvre s'appuient soit sur un protocole à base d'échange de messages, soit sur la possibilité d'accéder directement à l'ensemble des disques (cf. figure 2.1). En effet, certains protocoles intégrés aux baies de disques actuelles (*SCSI*, fibre optique) permettent à plusieurs machines d'accéder en concurrence aux diverses données stockées. les approches construites sur ces genres d'interface sont caractérisées par un modèle à accès direct[5]. Les critères de performances (débit, portabilité, granularité de partage, ...) sont donc considérablement influencés par l'approche choisie et les caractéristiques inhérentes aux matériels employés.

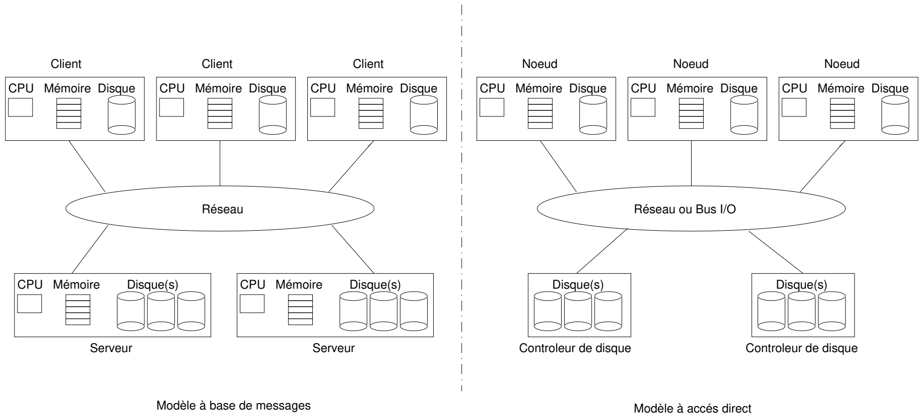

FIG. 2.1 – Protocole de partage de données

A coté de ces optimisations, les systèmes de fichiers «nouvelle-génération» fournissent généralement une A.P.I. spécifique à la programmation *I/O* parallèle plus ou moins élaborée (*Collective I/O*, accès diffus, ...). Ces routines, qui viennent parfaire dans la majorité des cas les opérations POSIX usuelles, sont pour la plus part très compétitive mais reste foncièrement dépendante au modèle déployé. La portabilité est un

[3]Plusieurs systèmes de fichiers distribués [HO93, ADN+95, PBB+00, Sch02, LD02] susceptibles d'être utilisés au sein des *clusters* sont présentés dans [Leb02].

[4]*Abstract Device Input/Output*

[5]En anglais : *Shared-Disk File System*.

critère qui fait relativement défaut à ce type de solutions.

2.2.1 De *Vesta* à *PIOFS*

Vesta

Vesta Parallel File System [CF01] est un système de fichier expérimental conçu par IBM en 1993. Il a été développé dans le but de fournir une interface d'accès adéquate aux applications s'exécutant sur des machines multiprocesseurs. Il repose sur une approche client / serveur (modèle à base de message entre des nœuds de calculs et nœuds de stockages).

La première notion importante relative à ce système est le placement physique des données sur une base 2D dynamique : un fichier est divisé en un nombre de cellules (partitionnement physique) spécifié lors de sa création ; chacune de ces cellules étant elle-même subdivisée en une taille également spécifiée d'unité basique (comparable aux blocs disques, cf 1.1.6). Les tailles de répartition des données étant positionnées lors de la création, il est alors possible de choisir pour chacun des fichiers une granularité permettant de minimiser les accès disques et de ce fait, obtenir de meilleures gains. Le second concept est la mise à disponibilité d'une interface permettant de créer des vues logiques pour accéder aux données. Cette notion, entièrement reprise par la suite au sein de *MPI I/O* , offre au programmeur un accès transparent à des informations plus ou moins diffuses (les noms «béguin» ou «partitionnement logique» sont employés dans la terminologie spécifique au modèle *Vesta*.

L'exploitation de module de cache intégré aux serveurs d'entrées/sorties permet d'optimiser les accès en lecture et en écriture. La cohérence est alors maintenu au sein du système via un algorithme à base de jetons[6]. Les opérations d'écriture réparties sur plusieurs serveurs de stockages reposent également sur cette méthode afin d'assurer la consistance requise.

Vesta est un système qui a pris en compte plusieurs exigences de la programmation parallèle *I/O* intensive (la gestion des fichiers supérieur à 2Go est également fournit). Néanmoins, il a été choisi de ne pas intégrer l'interface POSIX ce qui a réduit considérablement son essor. De même, aucune optimisation au niveau des opérations collectives n'est incluse. La gestion du cache du coté client est totalement absente (les accès de type Lecture cf 1.1.3 ne peuvent être capitalisés). Enfin, un inconvénient majeur concerne également la tolérance aux pannes : le protocole d'échanges de données requiert la présence de toutes les entités pour fonctionner correctement et aucune politique de protection n'a été mise en place.

Parallel I/O File System

Le système PIOFS [CFP+95] est une continuité des travaux entrepris lors du projet *Vesta*. Intégré au système d'exploitation *AIX* d'IBM, il fournit en plus des fonctionnalités proposées par le modèle précédent, une partie de l'interface POSIX[7] indisponible auparavant. Toutefois, cet apport primordial n'a pas réellement accru le coefficient de portabilité du modèle. En effet, l'appel aux routines optimisées d'accès parallèles reste explicite (et par conséquent figure toujours au sein du programme).

[6]Seul le processus qui a le jeton peut exécuter une opération.

[7]Une couche permettant d'abstraire l'interface *Vesta* a été ajoutée au noyau AIX.

Un des avantages certain de *PIOFS* est la mise à disposition d'outils permettant le suivi des applications. Ces fonctionnalités (trace et *Profilling*) se révèle très utile dans le cadre du développement d'applications scientifiques hautement parallèles[8].

2.2.2 *Parallel Input/OUtput System*

L'architecture *PIOUS*[9], présentée courant 1993, a introduit le concept de groupe au sein des opérations d'entrées/sorties parallèles [MS94]. En plus des vues logiques, nommées *para-fichiers* et comparable aux cellules dans *Vesta*, il suggère trois modes d'accès parallèle au sein d'un ensemble spécifié de nœuds (notion de groupe partageant une même vue logique d'une ressource) :

1. vue globale, le *para-fichier* apparaît comme une séquence linéaire d'octets. Chaque unité concernée partage un pointeur commun (cf. paragraphe 1.1.3) sur cette ressource (toute opération est atomique)
2. vue indépendante, à la différence du mode global, un pointeur est attribué à chaque processus.
3. vue segmentée, chaque nœud du groupe accède de manière atomique à un ou plusieurs segments indépendants du *para-fichier*.

Le modèle *PIOUS* fournit également un service de verrou avec une granularité correspondant à l'octet (évitant ainsi les problèmes de «faux-partage», cf 1.1.6). Un système de transaction (analogue à celui utilisé dans les *SGBD*s[10]) est proposé au sein de l'interface afin de permettre une programmation rigoureuse si nécessaire (cf 1.5, atomicité des opérations).

Vue d'ensemble, l'architecture se compose d'un service de coordination (*PSC*) contenant les méta-informations et l'état du système, d'un ensemble de serveurs de données (*PDS*) et d'une librairie de routines d'accès parallèles à lier avec les applications clientes. La particularité de ce système est l'utilisation de systèmes sous-jacent. Les serveurs (*PSC* et *PDC*) s'appuient sur des systèmes de fichiers natifs (*UNIX*, *XFS*, ...) pour stocker de manière pérenne les données. Par ailleurs, un mécanisme de transport pour l'échange des messages entre les entités est requis ; dans la plus part des implémentations, *PIOUS* repose sur l'intergiciel *PVM*[11].

2.2.3 *Galley File System*

Développé au collège Darthmouth à la suite du projet *CHARISMA*, [NK97], il a eu pour principal but la mise à disposition d'un système proposant des routines d'accès parallèles adéquates à l'ensemble des applications *HPC* . L'idée majeure est de concevoir une architecture capable de s'adapter à une grande diversité de librairies de haut-niveau plutôt qu'essayer de prévoir la totalité des caractéristiques impératives à un logiciel scientifique. Le système de fichiers est complété par une couche «métier» optimisée et spécifique aux exigences émises.

Une des toutes premières distinctions du système *Galley* part du constat qu'une distribution statique des fichiers sur n unités de stockages peut se révéler moins perfor-

[8] Plusieurs équipes de recherches concentrent leurs efforts sur ces outils permettant le suivi des opérations d'entrée/sortie. Parmi les plus connues, nous citerons le projet *Pablo* (cf. *PPFS*).

[9] `http://www.mathcs.emory.edu/pious/`

[10] Système de Gestion de Base de Données.

[11] *PVM* permet d'abstraire un *cluster* en une unique machine virtuel.

mante que l'approche linéaire des systèmes de fichiers standards[12]. Ainsi la répartition des données sur l'ensemble des unités de stockage, appelé *IOP* n'est pas imposée (tout fichier de taille quelconque peut être sauvegarder sur un unique disque).

S'appuyant sur une approche client / serveur (unité de calcul *CP* et nœud de stockage *IOP*), les processus de calculs sont indépendants entre-eux et ne nécessitent pas un intergiciel de communication évoluée de type *PVM* ou *MPI* comme il est requis au sein de *PIOFS*. S'appuyant sur le multi-tâche et les couches TCP/IP, il intègre les opérations asynchrones le cas échéant et utilise un gestionnaire de cache sur les serveurs de données afin d'optimiser les opérations d'entrées/sorties. Un gestionnaire de disque permet d'interfacer le système de fichiers natif installé sur l'*IOP*.

Une fois que le client a chargé la libraire *Galley*, il peut utiliser le système de partage de données. En plus des traditionnelles routines *UNIX*, la libraire implémente des appels plus complexes qui fournissent un contrôle maximal sur le placement physique des données lors de la création des fichiers mais également la possibilité de réaliser des accès enchevêtrés (cf 1.1.5).

Une modélisation proche de celles présentées dans les architectures précédentes est mise en place afin d'abstraire les fichiers et de proposer des vues logiques :
ces derniers sont composés de un ou plusieurs «sous-fichier(s)» sauvegardé(s) chacun sur une unité de stockage à part. Chaque sous fichier est constitué d'un ensemble de taille variable de «*fork*» (composant comparable à un fichier traditionnel *unix* mais moins «onéreux» en terme de gestion). Galley introduit de cette manière une visualisation tri-dimensionnelle d'un fichier ce qui permet d'apporter un niveau de sécurité supplémentaire de même qu'un stockage plus discret des données et méta-données correspondantes [13].

A ce jour et à ma connaissance, seules les interfaces des systèmes *Panda* [Tho96] et *Vesta* [CK98] ont été implémentés au dessus dessus de l'architecture *Galley*. En effet, l'interface complète fournit par le modèle *Galley* est quelque peut complexe pour un programmeur qui souhaiterait directement exploiter les routines proposées.

2.2.4 *Parallel Virtual File System*

Le système de fichier *PVFS* a été introduit courant 1996, [LR96]. Il a pour principal objectif l'exploitation optimal des ressources présentes au sein des *clusters beowulf* (bande passante, espace disque disponible, ...). Basé sur une architecture client / serveur, une fois encore, le système se décompose en un méta-serveur (*mgr*, stockant les méta-informations) et en un groupe de serveur de données (*iods*, pour les données effectives).
Chaque fichier est découpé en blocs de taille fixe répartis de manière circulaire sur l'ensemble des unités de sauvegarde. Lorsqu'une application désire lire ou écrire des données, les informations concernant la distribution sont d'abord récupérées auprès du manager par la machine cliente. Une fois le découpage connu, les clients accèdent directement aux serveurs de données. Les problèmes conséquents au goulot d'étranglement au niveau du méta-serveur sont ainsi largement diminués puisque seules les

[12] En effet, les requêtes de taille supérieure au coefficient de répartition génèrent une gestion plus complexe et une latence I/O plus élevée.

[13] Il est, par exemple, impossible de dissocier les méta des données ou encore d'effacer malencontreusement les méta-fichiers.

méta-requêtes «stressent» le serveur[14].
Un principe fondamentale du modèle est l'agrégation des requêtes *I/O* au sein d'un même trame réseau (*Stream-Based I/O*). Cette technique, comme nous l'avons indiqué au paragraphe 1.2.2, permet de réduire considérablement les flux réseaux et éviter ainsi les phénomènes de contention.

A l'instar de *PIOUS* ou bien *Galley*, les serveurs constituant *PVFS* s'appuient sur les systèmes de fichiers locaux et bénéficient par conséquent, des politiques de caches déjà présentes au sein des architectures. Il est important de préciser que ces stratégies de caches ne sont pas implémentées du coté du client ; la mise en cohérence du système est ainsi facilité, l'ensemble des opérations étant synchrones.

L'environnement *PVFS* propose trois interfaces d'accès. Les routines POSIX sont fournit en utilisant la couche *VFS*[15]. Une librairie pour les opérations sur des données diffuses vient parfaire l'interface UNIX (définitions de vues logiques,*MDBI*[16]). Enfin, une implémentation d'un pilote pour l'*ADIO* de *MPI I/O* a été réalisé au sein de *ROMIO*. Une paragraphe y est consacrée au sein du chapitre **??**.

Le modèle *PVFS* a relativement été accepté par la communauté *HPC*. A ce jour, c'est un des systèmes le plus déployé au sein des architectures de type *cluster* . Plusieurs travaux ont suggérés diverses optimisations ([VKS02], par exemple, propose un module noyau intégrant des services de cache). Toutefois, l'absence de certaines fonctionnalités fondamentales comme la tolérance aux pannes (dépendance entre les serveurs de données), l'inexistence des verrous ou encore la distribution figée des informations ne permettent pas une implantation unanime du modèle.

Depuis 2002, un squelette, *PVFS 2* est en cours d'étude afin de corriger et d'éliminer les divers inconvénients. Tout d'abord, l'ensemble des serveurs sera susceptible de gérer tout type de requêtes (le méta-serveur sera distribué sur l'ensemble des nœuds de stockage par un système de *hash*) afin d'éliminer toute présence de point critique. Les routines aux données non contiguës seront complétées afin de pouvoir accéder à des informations reposant sur des modèles à régularité double, triple, ...(cf paragraphe 1.1.5). Un module d'opérations asynchrone devrait également être fournit (les problématiques de cohérence devront par conséquent être étudiées). Pour terminer, comme l'a suggéré le projet *Galley*, la nouvelle structure permettra de lier rapidement des modules pour des caractéristiques spécifiques (distribution de données, ordonnancement, protocole de transports). Une interface commune permettra de centraliser l'ensemble des librairies spécifiques au type d'application mais également les pilotes sous-jacents. Le système *PVFS 2* pourra par ce moyen, être performant et plus facilement portable.

2.2.5 *Lustre*

Lustre[17], [whi02, Sch03], est une solution en court de développement (*Lustre Lite*) et a pour objectif l'implémantation d'un système de fichiers approprié pour les architectures de type *cluster* [18]. La brève description qui suit n'insistera pas réellement sur

[14] A la différence des modèles reposant sur le protocole NFS où la totalité des requêtes transite par le point centralisé.
[15] Ce module, connu sous le nom *Virtual File System*, est inclut dans la quasi-totalité des systèmes *UNIX*. Il permet d'abstraire en une unique interface plusieurs systèmes de fichiers.
[16] Multi Dimensional Block Interface
[17] `http://www.lustre.org`
[18] L'acronyme *Lustre* provient de l'association mots *Linux* et *cluster* .

les aspects parallèles et les techniques mises en œuvre afin d'en optimisés les accès (l'architecture étant encore très évolutive). Toutefois, le modèle agréant de nombreux concepts suggérés par la communauté scientifique depuis plus d'une quinzaine d'années[19], [Bra99], semble être très prometteur (partage à grain fin, fiabilité, portabilité, scalabilité, ...). Son déploiement sur plusieurs gros super-calculateurs ainsi que le soutien apportait par de multiples partenaires privées ou universitaires (*HP*,*Seagate*, Université *Carmegie Mellon*, *Lawrence Livermore National Lab*, ...) nous incitent à prendre en compte d'ores-et-déjà ce modèle afin d'observer comment les problématiques liées aux accès parallèles au sein des applications *HPC* pourront être organisées.

L'architecture est globalement comparable au modèle *PVFS*. Un méta-serveur, *MDS*, sauvegarde l'état du système de fichiers (les méta-informations) et des serveurs de données (*OST*, *Object Storage Target*) gèrent l'ensemble des requêtes *I/O* effective (interface avec les unités de stockages sous-jacentes). Afin de réduire les inconvénient engendrés par un point centralisé, le manager est répliqué (une image «miroir» est réalisée mais non activée) et intervient uniquement lors d'opérations concrètes sur les méta-informations (création, destruction et mise à jour).

Axé sur les concepts de système de fichiers sauvegardé sous forme d'objet (*object based disk*, cf. Annexe XX) offrant une meilleur exploitation des disques, *Lustre* fournit tout de même plusieurs pilotes permettant d'accéder à des systèmes de fichiers locaux comme *Ext3*, *ReiserFS* ou encore *XFS* et donner ainsi la possibilité au modèle de s'adapter à plusieurs types d'architectures.
Remarque : *Lustre* exploite également les méthodes de journalisation intégrées à ces systèmes de fichiers afin d'augmenter sa robustesse.

Une troisième entité vient compléter le modèle *Lustre* ; en effet un serveur indépendant de type *LDAP* détient les informations concernant la topologie de l'infrastructure. En cas de panne ou de dysfonctionnement (expiration d'un délai de garde), le nœud qui a perçu la défaillance interroge ce serveur «extérieur» afin de connaître l'unité de remplacement. La fiabilité du système est une nouvelle fois accru.

Une autre caractéristique de *Lustre* est son indépendance vis-à-vis du réseau et des protocoles utilisés grâce à l'utilisation de la bibliothèque *Portal*[20]. Une couche basse du modèle, *NAL* (*Network Abstraction Layer*), fournit une interface commune à de multiples protocoles réseaux. Actuellement, le système supporte les flux *TCP/IP*, les réseaux *Quadrics*, *Myrinet* ou encore *SCI* (d'autres pilotes, comme *InfiniBand* notamment, sont en phase de développement). Ce détachement permet au système tout d'abord de bénéficier des apports réalisés à tous moment au sein des protocoles réseaux mais par ailleurs de coupler ces différents supports au sein d'une même et unique architecture (certains clients accèdent aux différents serveurs par l'intermédiaire d'un réseau *TCP* et d'autres par un réseau *Quadrics* par exemple). Le protocole *Lustre* offre également la possibilité de «court-circuiter» les piles réseaux au cas ou un même nœud inclût le processus client et le processus *OST*.

D'un point de vue de la cohérence, le système comporte un module de verrous permettant un partage des données à grain fin tout. Le *DLM* (*Distributed Lock Manager*) intervient également pour maintenir les différents gestionnaires de caches intégrés au

[19]Le modèle est un descendant du système *VAX Cluster* et exploitent plusieurs notions également utilisés dans le plus récent *GPFS*.

[20]Conçu au *Sandia National Laboratory*, l'intergiciel *Portal* est une librairie d'échange de messages relativement simple. Sa particularité principale est son indépendance par rapport aux divers systèmes d'exploitations.

système. Dans un premier temps, les clients ont exploité un cache permettant de réaliser des écritures «retardées» sur les méta-informations. Cette couche a été étendu vers les concepts de cache persistant implémenté au sein de systèmes de fichiers comme *AFS*, *CODA* ou encore *Intermezzo*, [Leb02]. Donnant la possibilité de travailler en mode déconnecté dans les systèmes de fichiers précédemment cités, le module mis en place est utilisée ici afin de réduire les flux réseaux : un «méta-serveur local» est ajouté sur chaque client, une image cohérente des méta-informations est ainsi disponible ponctuellement. Cette image est propagée vers le *MDS* distant lors des requêtes ne pouvant être satisfaite localement («défaut de page»).
Parallèlement au gestionnaire de cache des méta-données, un gestion de cache «globale»[21] a été mis en place pour optimiser les accès directes vers les *OSTs*. Ces «serveurs» indépendants déployés sur des nœuds distants récupèrent une copie de la portion de données souhaités. Les clients qui émettront des accès vers ces données, seront redirigés par l'*OST* concerné, vers l'image temporairement stockée sur le gestionnaire distant. Cette approche réduit les phénomènes de saturation pouvant intervenir sur un serveur de stockage lorsqu'un trop grand nombre de clients tente d'accéder à des données présentes au sein d'un même *OST*.

Le système de fichiers *Lustre* est en phase de développement intensive depuis un peu plus d'un an. L'objectif à court terme est le passage à l'échelle à plus de 2000 nœuds. Une couche permettant d'interfacer la librairie *MPI I/O* est également en cours d'étude. Elle devrait être constituer d'un couche sous-jacente à l'*ADIO* du coté client et d'un couche de synchronisation des opérations collectives aux niveaux des *OSTs*.

2.3 Les librairies

Généralement moins intrusive d'un point de vue des systèmes d'exploitation, les librairies consistent en une collection de routines d'entrées/sorties parallèles utilisables par simple appel au sein du code. Ces «bibliothèques» composées de plusieurs couches logiciels peuvent rapidement être adaptées à une nouvelle architecture sous-jacente[22] et proposer, ainsi, un meilleur coefficient de portabilité aux applications *HPCs* qui les exploitent. Il est également plus facile d'accéder au sein des diverses couches afin d'apporter d'éventuelles modifications spécifiques à certains applications ou compléter les interfaces avec de nouvelles politiques d'optimisations.

Néanmoins, le développement ainsi que l'usage de librairies imposent une étude minutieuse, tout d'abord, des caractéristiques et des performances du système sur lequel elle repose (accès séquentiel vs parallèle, synchrone vs asynchrone, UNIX vs Windows NT, partagé vs dédié, granularité des unités de stockages, ...) mais également des couches intermédiaires qu'elles exploitent. En effet, construite sur des systèmes d'échanges de messages plus ou moins complexes (*TCP/IP*, *MPI* ou encore *PVM* ...) permettant les communications inter-nœuds, il est impératif lors de la mise en œuvre de connaître toutes éventuelles limitations pouvant être induite par ces sous modules (débits, scalabilité, opérations collectives ...). Plusieurs bibliothèques ont été suggérées ; nous avons répertorié les principales ci-après.

[21] Un module similaire est employé au sein de la libraire *PPFS*, cf. paragraphe 2.3.3.

[22] A défaut des systèmes de fichiers précédemment présentés, qui sont eux, étroitement liés aussi bien avec leurs couches logicielles qu'avec la configuration matériel exigée.

2.3.1 *PASSION*

Le programme *Parallel And Scalable Software for Input-Output* [CBH+94], débuté en 1994, a eu pour principal ambition la mise en place d'un système capable de résoudre les problèmes *I/O* dans les applications *out of core*[23] sur architectures à mémoires distribuées. Une des premières caractéristiques est la mise à disponibilité d'un support pour la compilation. Ainsi, *PASSION* fournit un module capable de compiler du code *Fortran 90D* et du *HPF* afin de réaliser certaines optimisations implicitement. Ces techniques (*prefetching*, *Data Sieving*, opération collective, ordonnancement optimisé des données[24] ...) sont également accessibles en phase d'exécution par l'intermédiaire de routines spécifiques inclues au sein de la libraire. S'appuyant sur un modèle scindé en plusieurs entités (gestionnaire de cache, gestionnaire d'opérations collectives, sous module d'I/O), *PASSION* peut interagir de manière optimale avec le système de fichiers sous-jacent en exploitant à chaque niveau la connaissance du placement des données. Par ailleurs, de nombreuses fonctions permettant la gestion des calculs *out-of-core* (multiplication de matrices, transformations ...) sont inclues au sein de la bibliothèque.

D'un point vue matériel, chaque entité composant le *cluster* peut se comporter comme un nœud de calcul, d'*I/O* ou les deux également. Le système est indépendant à la topologie réseau utilisée (partage du réseau principale vs réseau *I/O* dédié, exploitation de disques locaux ou de baies de stockages spécifiques). Les couches *PASSION* sont utilisables au dessus des systèmes de fichiers Intel CFS et *Vesta*.

Parallèlement, le projet *PASSION* a développé un système de fichier nommé *VIP-FS*[25]. Ce nouveau modèle de partage de données est implémenté au dessus de plusieurs systèmes de fichiers et utilise un interface d'échanges de messages afin de coordonner les opérations (cette solution possède donc le même degré de portabilité que l'interface de communications sur lequel il repose). L'architecture totalement distribuée, est divisée en deux couches[26] : une de haut niveau,*PFI*, *parallel File Interface* et une, *LDI*, *Local Device Interface* permettant l'interaction avec le système de fichiers natifs. L'emplacement physique des deux modules n'est pas primordiale, ils peuvent être déployés sur un même nœud aussi bien que sur deux machines distantes.
Le système est exploitable par l'intermédiaire des routines de haut niveau alors que le composant *LDI* effectue uniquement la translation de ces appels vers les systèmes de fichiers sous-jacents. La seul exception apparaît lors des opérations collectives ou un unique *PFI* élu parmi les nœuds composant le *cluster* transmet la requête collective vers tout les *LDIs* (cette méthode est comparable à l'approche décrite au sein du paragraphe 1.2.3, *disk directed I/O*). Chaque *LDI* retourne les informations qu'il détient au nœud de calculs concerné.

Une caractéristique intéressant de ce système de partage de données est sa dynamicité vis à vis à la plate-forme matérielle sur laquelle il est déployé. Ainsi, il est possible pour chaque application de choisir la composition matérielle optimale du système (nombre de nœuds I/O / unité de calculs / Débit maximum réseau). Dans le cas où l'application laisse le libre arbitre, *VIP-FS* s'adapte à l'architecture sous-jacente en recherchant les nœuds susceptibles de fonctionner en serveur d'entrées/sorties.

[23]Les logiciels scientifiques basés sur une grande quantité d'informations ne pouvant toute être contenue au sein de la mémoire principale.
[24]*Data Reuse*.
[25]*VIrtual Parallel File System*.
[26]Nous verrons que la même approche a été utilisée au sein de *MPI I/O* .

2.3.2 *PANDA*

Introduit en 1994, le projet *PANDA* [SCJ+95] s'est concentré sur les problèmes *I/O* attenant à la gestion des tableaux à plusieurs dimensions. En 1995, cette librairie intègre les opérations collectives en s'appuyant sur une abstraction du modèle proposé par [Kot01] permettant une meilleur utilisation des débits offert par les unités de stockages (cf. paragraphe 1.2.3). Egalement fondée sur une architecture client / serveur développée en C++, *PANDA* s'appuie sur la libraire d'échange de messages *MPI* et sur tous systèmes de fichiers proposant une interface UNIX.
Cette bibliothèque maximise les accès séquentiels dans le but d'améliorer les performances. Elle procure la possibilité de distribuer les informations au sein des diverses serveurs de stockages selon une disposition donnée. D'une manière générale, le placement physique des données correspond à celui employé en mémoire (*natural chunking*).

Une opération collective se déroule de la manière suivante : un client, élu (*Master client*) parmi le groupe concerné par l'opération collective, transmet vers un serveur également choisi (*Master server*) une description de la disposition des données en mémoire et du placement souhaité sur les disques. Les serveurs de stockages vont exploiter ces informations afin de transmettre ou récupérer auprès des clients chaque(s) bloc(s) nécessaire à l'exécution de la requête. Lorsque l'opération I/O se termine sur un serveur, celui-ci en averti le gestionnaire de la requête (*Master server*). Une fois que la totalité des serveurs de stockages ont «accusé» l'opération, le gestionnaire en fait part au client principal. Ce dernier indiquera le bon déroulement de l'opération à l'ensemble des nœuds clients.

De nombreuses fonctionnalités ont été ajoutées depuis la création du projet. En 1997, la librairie *PANDA* fournit une interface permettant la migration performante de données entre deux architectures[27] durant l'exécution d'un programme [KWC+97]. Par ailleurs, l'architecture devient plus malléable ; en effet, l'usage de serveurs *I/O* dédiés ne devient plus obligatoire. Un sous-module, *Part-time I/O* [CWS+97], est inclu afin de court-circuiter le processus et utiliser de manière optimale le système de fichier local (suppression des latences réseau dues à l'échange de messages client / serveur). Cette étendue offre également la possibilité de faire varier le nombre de serveur *I/O* tout au long du déroulement du programme. Par ce moyen, il est envisageable de rééquilibrer le système *PANDA* (nombre de serveurs *I/O* déployés) afin d'optimiser les performances. En 2002, il intègre une nouvelle optimisation au sein des opérations collectives [MWLY02]. la technique *active buffering* rend transparent les latences *I/O* aux lieux de les réduire. Dans le cadre d'opération d'écritures collectives, l'espace mémoire disponible sur chaque serveur est utilisé afin de sauvegarder temporairement les données transmises par les clients. Une fois que chaque serveur a récupéré les données au sein de ces tampons temporaires, il valide l'opération au serveur maître qui transmettra l'information aux clients. L'opération effective de l'écriture des données sur disques sera alors réalisée lorsque les serveurs *I/O* seront inactifs (aucune requête est en attente sur le serveur). Cette méthode a été récemment proposée au sein de implémentation *ROMIO*.

Le projet PANDA est toujours actif. Intégré au sein d'un projet de simulation de fusée, la bibliothèque *RocPANDA* [MJoMW03], se concentre sur les problématiques liées à la compatibilité des données et des accès entre les différentes modules composant une application *HPC* (modules en permanente évolution et qui de plus sont

[27] Par exemple, d'un *cluster* vers une station d'analyse indépendante de l'architecture.

développés par plusieurs groupes de recherches souvent indépendant et programmant sous des langages et selon des règles différentes *C*, *C++*, *HPF*, ...).

2.3.3 *PPFS*

Portable Parallel File System [HER+95] est un système développé au sein de l'équipe *PABLO*[28]. Fondée sur une approche client /serveur, cette libraire permet de contrôler d'une part le placement des données sur les diverses unités de stockages mais également les diverses stratégies d'optimisation implémentées.
Ainsi, *PPFS* fournit :

une gestion du cache paramétrable, de manière global, au niveau des clients, des serveurs, choix de la granularité, écriture retardée ;

des techniques de *prefetching,* sur les clients et les serveurs, dans le but de minimiser au maximum les latences *I/O* ;

un degré de cohérence, plus ou moins forte entre les méta-informations mais également au sein des données ;

plusieurs modes d'accès, séquentiel, aléatoire, bloc-séquentiel, structuré pour les données diffuses.

Le modèle *PPFS* s'appuie sur tout type d'architecture parallèle incluant une libraire de communication, la plupart du temps *MPI*, et proposant un système de fichiers POSIX. Les nœuds de calculs utilisent la partie «cliente» de *PPFS* en appelant les routines adéquates au sein de leur code.

Chaque fichier est divisé en bloc de taille variable réparti en segments distribués séparément sur l'ensemble des serveurs de données. Le méta-fichier contient la totalité des informations permettant l'optimisation des accès, à savoir : la répartition des segments, la taille des blocs, les modes d'accès, ... Le méta-serveur intervient lors des opérations d'ouverture et de fermeture et maintient les méta-informations en cohérence. Les clients sont susceptibles de se transmettre les méta-informations afin d'éviter un éventuel goulet d'étranglement pouvant survenir sur le méta-serveur.

Comme nous l'avons précédemment indiqué, *PPFS* exploite les techniques de caches à plusieurs niveaux. Le concept d'agent de cache consistant en une unité globale par laquelle les clients peuvent se partager les données d'un fichier offre une répartition des demandes *I/O* entre agents et serveurs de données. La couche «cliente», susceptible également d'intégrer un cache local, interroge le «méta-serveur» afin de déterminer quels serveurs de données ou agents de caches va pouvoir satisfaire sa demande. Toutes les requêtes futures sur ce fichier seront transmises directement vers l'entité spécifiée. L'utilisation des ces «agents» est dynamique : des routines destinées à la mise en place ainsi que la suppression de tels modules sont inclues au sein de la bibliothèque *PPFS*. Il est, par conséquent, possible de gérer de manière optimale les charges *I/O* durant tout le déroulement de l'application *HPC* . Une autre point intéressant de ce concept est une gestion «paramétrable» de la cohérence. En effet, lorsqu'un agent est instancié, la totalité des accès au fichier associé transite par ce point centralisé et pourra être maintenue en cohérence (l'utilisation de cette méthode s'avère moins coûteuse que la mise en place de cache distribuée).

[28]Un des thématiques principales de ce laboratoire est l'évaluation et l'optimisation des architectures parallèles (`http://www-pablo.cs.uiuc.edu/`).

Développé via les concepts objets (C++), *PPFS* peut être aisément enrichi afin d'étudier des nouvelles approches. Proposant des routines de placement et de gestion à chaque étape du processus d'accès aux données, il est possible d'implémenter une «sur couche» héritant des caractéristiques inclues au sein de *PPFS* dans le but de valider une stratégie novatrice.
Récemment, l'équipe *PABLO* a présenté un nouveau modèle : *PPFS II* exploite un logiciel permettant d'analyser en «temps-réel» l'état et les performances de l'architecture sur lequel il est déployé. Les résultats fournis par ce module, nommée *autopilot* est développé également par *PABLO*, sont utilisés par la librairie en vue de réajuster de manière dynamique et transparente la stratégie d'optimisation employée. L'équipe *PABLO* est très active et continue de proposer plusieurs outils (optimisation, suivi et traces des entrées/sorties, ...).

2.3.4 *Jovian*

En 1994, dans le cadre du projet *CHAOS*[29], la librairie *Jovian*, [BBS$^+$94], a été présentée. Cette bibliothèque permet d'optimiser les accès *I/O* au sein des programmes *SPMD*[30] déployés sur des architectures composées de plusieurs unités de stockages. Le concept fondamentale est que toute opération *I/O* se déroule de manière collective (tous les nœuds communs à l'application réaliseront l'opération et ceci dans tous les cas).

Le modèle employé pour implémenter *Jovian* repose sur deux entités principales : les processus de calculs communément appelés *APs* pour *Application processes* et les processus en charge des opérations d'entrée/sortie, les *CPs* pour *Coalesce processes*. Lors de la phase d'initialisation du programme, la librairie *Jovian* instancie les processus *APs* et *CPs* via les informations de configurations fournies par l'utilisateur. Un processus *AP* est alors attribué auprès d'un processus *CP*. La durée de vie des processus *CPs*, comparables alors à des serveurs d'accès aux données, est corrélée à l'exécution de l'application. Les *CPs* qui sont construis au dessus d'un ou plusieurs systèmes de fichiers sous-jacents gèrent chacun une partie physique du fichier (grossièrement taille du fichier/nombre de *CPs*).

Lors d'une opération *I/O* de type lecture ou écriture, chaque *AP* émet de manière bloquante, sa requête vers un *CP* déterminé lors de la phase d'initialisation. Dès cette première étape, la librairie, coté client, exploite une méthode analogue à l'algorithme *Data Sieving* largement décrite au chapitre précèdent. L'intérêt est de favoriser les échanges réseaux en limitant les phénomènes de contentions pouvant subvenir durant toute l'exécution de l'application. A la réception des demandes, l'ensemble des *CPs* entre dans une phase de communication collective afin de redistribuer, suivant la portion de fichier requis, les demandes *I/O* au *CP* en charge de cette plage d'octets (attribution des requêtes logiques par rapport au placement physique). Une fois que la totalité des demandes a été correctement redirigée, chaque *CP* exécute de manière similaire aux *APs*, une méthode d'agrégation des requêtes (*Data Sieving*) afin de maximiser l'utilisation de la bande *I/O* disponible[31]. L'accès «physique» est alors réalisé sur l'unité de stockage correspondante soit par l'intermédiaire d'un contrôleur permettant d'interfacer directement les baies de disques ou simplement en s'appuyant sur l'interface POSIX du système de fichiers natif. Les données récupérées sont placés au sein

[29] Université du maryland.
[30] *Single Program Multiple Data*.
[31] Le paramètre *Read Gap Size* est utilisé afin d'éviter les agrégations trop coûteuses, cf 1.2.4.

de tampons «d'envoi» (un tampon d'envoi par *AP* concerné) ; ces derniers seront transmis lorsque l'opération «physique» sera terminée (ou que le tampon temporaire sera saturé). Lors des opérations d'écritures, une approche *read-write-modified* est mise en œuvre durant cette dernière étape afin d'éviter les problèmes de «faux-partage».

Une seconde version de la libraire *Jovian*, [AUB$^+$96], a été mise en place par la suite. Fondamentalement différente, *Jovian 2* est une version multi-tâches de la librairie *Jovian*. Même si elle continue à exploiter un modèle fondé sur deux entités : un processus client et un processus serveur, les concepts de base sont différents et cette nouvelle bibliothèque ce distingue sous quatre aspects majeurs :

1. *Jovian* s'appuyait sur un approche synchrone des opérations *I/O* : seul des opérations collectives pouvaient apparaître. La nouvelle version a supprimé la totalité des routines collectives et propose uniquement une interface POSIX simple permettant par l'intermédiaire de la routine $lio_listio()$ d'accéder à des données non contiguës en un unique appel.
2. Comme, il a été indiqué l'approche employée au sein de *Jovian* consistait en une agrégation des requêtes de manière collective. Ainsi, toute opération d'entrée/sortie ne pouvait physiquement être exécutée qu'une fois l'ensemble des «pré-phases» terminées. A la différence, *Jovian 2* tente de maximiser l'utilisation du disque en retournant de la manière la plus rapide la plage de données demandée.
3. Le module *CP* est implémenté au sein d'un *thread* intégré soit à l'application cliente soit sur un nœud distant. L'intérêt est de pouvoir réaliser des architectures «pair à pair» si besoin est. Dans ce cas tous nœuds clients est susceptible de se comporter comme un serveur pour tout autre machine. Ainsi il est également possible de «court-circuiter» la procédure d'accès aux *I/O* pour toute demande à destination du *thread* local et éviter les étapes de communications (technique notamment employée au sein de la librairie *PANDA* et repris lors de la conception du système de fichier *Lustre*).
4. Enfin, cette solution offre la possibilité de gérer la distribution du placement physique des données sur les unités de stockages sous-jacent (la répartition simpliste utilisée précédemment devient «configurable»).

2.3.5 *MTIO*

Cette librairie [MCFX97], conçu en 1997, s'appuie également sur les concepts multi-tâches dans le but de réduire les latences *I/O* et l'exécution des opérations collectives. Elle s'attarde principalement sur l'optimisation des accès individuels ou collectifs. Dans le cadre des requêtes indépendantes, elle exploite le *multi-threading* afin de maximiser l'utilisation du processus. Lors du lancement du programme, chaque processus, créé sur un nœud, se compose d'un premier *thread* de calcul (*CT*) et d'un second pour les opération d'entrée/sortie (*IOT*). L'idée est de réaliser l'appel *I/O* (souvent bloquant sur la quasi-totalité des systèmes de fichiers) dans le processus léger *I/O* et de pouvoir ainsi exploiter le *CPU* pour exécuter une autre partie du code en attendant les informations retournées une fois la demande exécutée.

La gestion des accès, de la cohérence et du partage des données doit être réalisées par le système de fichiers sous-jacent. La libraire *MTIO* qui emploi notamment la couche MPI pour la communication inter-nœud, fournit uniquement deux modèles d'accès : accès individuel, décrit succinctement auparavant et un modèle collectif. La spécificité de ce second mode est la prise en compte de paramètres permettant dans un premier temps, de minimiser la phase de communication inter-processus (phase relativement coûteuse au sein de l'algorithme *Two-Phase*) puis par la suite d'équilibrer au mieux les requêtes sur l'ensemble des nœuds concernés par l'opération collective. Lors de la première étape, les requêtes sont attribuées selon la localité des données d'un point de vue physique puis d'un point de vue logique. Lorsqu'une requête ne répond spécifiquement pas à une des critères de localité, elle est alors attribuée au *thread I/O* qui est pourvu du plus petit nombres d'appels. Un équilibrage des demandes constituant l'opération collective est ainsi fait.

Le projet *MTIO* ne semble pas avoir donnée suite. Toutefois, les problématiques soulevés dans le cadre de ce travail ont été réutilises dans de nombreux travaux.

2.4 Cohésion et portabilité des interfaces : *ADIO*[32]

Dans un souci d'uniformisation des interfaces et ne voyant aucune solution s'imposer comme standard, une équipe du laboratoire d'Argonne a proposé en 1996, une stratégie capable d'apporter un niveau de cohésion entre les diverses plate-formes et les multiples librairies d'entrées/sorties existantes. Ne pouvant s'appuyer ni sur l'interface usuel UNIX (section 2 [TGL99b]) trop restreint pour répondre a l'ensemble des contraintes des applications modernes ou encore sur le standard POSIX, certes plus complet, mais qui reste peu représenté dans le monde réel, [TGL96] présente un dérivée a partir des concepts éprouvés de l'*ADI* mises en œuvre au sein de l'implémentation *MPICH* du modèle *MPI* (cf. paragraphe 3.1.3). Consacré aux entrées/sorties : l'*ADIO*, ne consiste en aucun cas en une librairie supplémentaire et concurrente aux multiples solutions déjà présentes, il doit être simplement vue comme un «intermédiaire» (cf. figure 2.4) entre les libraires de haut niveaux et les systèmes de stockages généralement dépendant des caractéristiques physiques de l'architecture sous-jacente.

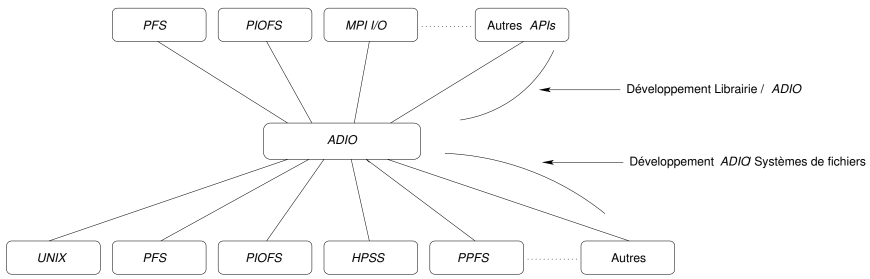

FIG. 2.2 – Concept du module *ADIO*
Toute interface d'entrées/sorties peut être implantée au dessus de tout systèmes de stockages en mettant en place la librairie au dessus de la couche *ADIO* et en développant l'*ADIO* correspondant à chaque architecture.

Disponible sur toute architecture *MPI*, l'*ADIO* a pour but de rendre les *API* de haut niveaux portables sans pour autant en altérer les performances. Une fois qu'une librairie

[32]*Abstract Device Interface for Parallel I/O*.

I/O a été implémenté au dessus du module, il est possible d'observer son comportement sur l'ensemble des structures de stockages possédant une jonction[33] avec l'*ADIO*.

Les routines présentes au sein de l'*ADIO* répondent à une syntaxe relativement semblable à celle du standard *MPI*. L'implantation des caractéristiques de chaque librairie (*Passion*,*Panda*,...) ou encore de chaque *API* (PIOFS, PIOUS,...) au dessus du modèle est rendu possible grâce à la richesse du standard *MPI* (notamment les types dérivées pour la correspondance entre vue logique et disposition physique). Parallèlement, l'implantation de module *ADIO* au dessus d'unités de stockages est facilitée par l'accroissement des fonctionnalités fournies par les systèmes de fichiers modernes (asynchronisme, accès diffus, ...). Dans un tel cas, l'interfaçage *ADIO*/Système de fichiers peut se résoudre parfois par un simple appel de routine. Cependant, il est nécessaire pour certains systèmes moins complet d'implémenter les fonctionnalités au sein des couches internes au modèle *ADIO* (*threads* pour l'asynchronisme agglomération des accès, ...).

Tout au long de ce chapitre, nous avons présenté plus ou moins succinctement les solutions majeures proposées à la communauté scientifique durant les années 90. Chacun de ces systèmes limitent les phénomènes d'étranglement *I/O* en réduisant l'écart de performances entre les unités de stockages et les autres composants d'une architecture informatique (*CPU*, bus, support réseau, ...). Toutefois, la grande disparité aux niveaux des interfaces de programmation ainsi que la multipléxité des méthodes employées affaiblit considérablement la portabilité des applications *HPC* . Les concepts soulevés par le modèle *ADIO* se révèlent être capitale pour les solutions postérieures. Nous le verrons notamment au sein de l'implantation *ROMIO* du standard *MPI I/O* qui repose entièrement sur ce modèle(cf. paragraphe 3.3.5). Nous y décrirons avec plus de précisions les différentes routines proposées au sein de la couche *ADIO* et examineront les techniques employées au sein de chacune.

[33] Le module d'intéraction *ADIO*/système de fichiers est acquis dès que ce dernier a été mis en place ; il en est également de même pour les interfaces de haut-niveaux.

Deuxième partie

MPI IO

Chapitre 3

MPI I/O

Le fort développement d'applications parallèles depuis une quinzaine d'années a provoqué l'émergence d'un très grand nombre d'environnement de programmation parallèle. Bibliothèques de partage de données comme *Cilk* [Gro01] ou *Open MP* [1], librairies d'échanges de messages comme *PVM* [GAD+94] ou encore l'ensemble des précurseurs de *MPI* ; ces outils proposent des fonctionnalités offrant au programmeur un développement plus souple et moins contraignant sur les architectures matériellement distribuées[2].
Dans le but d'une standardisation des routines employées et afin d'améliorer la portabilité des applications scientifiques, les utilisateurs et les industriels ont entrepris, en 1992, la conception d'une interface commune nommée *MPI* (*Message Passing Interface*). Le consortium *MPI* finalisa en Mai 1994 une première version du modèle puis décida en 1995 d'étendre ces spécifications vers de nouvelles fonctionnalités[3] (gestion dynamique des processus, communications inter groupes, temps réels, ...). C'est lors de cette seconde étape que le modèle *MPI I/O* , [CFF+01], a été introduit. Ainsi, une partie[4] des spécifications du standard *MPI-2* définissent un ensemble de routines permettant le transfert de données «efficace» depuis et vers les unités de stockages secondaires

Après avoir effectué une courte description de l'approche *MPI*, nous allons nous attarder sur la couche *MPI I/O* et ses diverses implémentations. Nous nous arrêterons plus précisément sur la solution *ROMIO* qui est mise en œuvre dans les deux architectures les plus représentatives du monde *MPI* à savoir *LAM*[SL03] et *MPICH*[GLDS96]. Nous verrons notamment que les analyses et remarques apportées par un grand nombre de travaux scientifiques ne cessent de faire évoluer le sous-module *ROMIO*.

3.1 *Message Passing Interface*

L'introduction à *MPI I/O* requiert une compréhension des concepts fondamentaux du modèle *MPI*. Ainsi même si, nous supposons que le lecteur est familiarisé avec

[1] `http://www.openmp.org`.

[2] Concept de «machines virtuelles» reposant sur une librairie (plus ou moins fournit) et d'un noyau exécutif.

[3] Le rapport regroupant l'ensemble des spécifications est disponible à l'adresse suivante : `http://www.mpi-forum.org/docs/docs.html`.

[4] Le chapitre 9 du standard *MPI-2* y est totalement consacré.

certaines notions basiques de *MPI*, nous avons choisi de rappeler très brièvement les caractéristiques ainsi que la structure utilisée afin de faciliter la compréhension des couches *MPI I/O* dans un second temps.

3.1.1 Généralités

MPI, comme son nom l'indique, consiste en une interface pour le passage de messages. Correspondant au modèle de programmation *SPMD, Single Program Multiple Data*[5], il est régi par des spécifications qui déterminent le comportement et les caractéristiques de chaque routine éliminant de ce fait les problèmes de portabilité vis à vis de l'architecture matérielle sous-jacente (seul les couches basses du modèle *MPI* doivent s'adapter aux ressources matérielles, l'application, développée à l'aide des routines spécifiques, peut être exécutée sur toute architecture répondant aux spécifications *MPI*). Le modèle inclut aussi bien des fonctions d'échange point à point que des méthodes collectives (mises en œuvre de primitives *send/receive* de type *1 à 1*, *1 à n* et inversement [6] ou encore *n à n*). Dans le but de «cibler» les opérations d'échanges, la librairie identifie de manière unique chaque processus instancié au sein de l'application (soit par un rang unique soit par l'intermédiaire de coordonnées spatiales correspondant à une topologie préalablement définie). Le modèle intègre également de nombreuses routines permettant l'encapsulation de données basiques ou plus complexes au sein de types *MPI* «portables». Il est ainsi possible d'échanger toute structure d'informations plus ou moins élaborée[7].

Parallèlement à l'*API MPI*, de multiples fonctionnalités sont venues se juxtaposer afin d'exploiter au mieux l'architecture matérielle sous-jacente (module d'investigation de performances matérielles, système de traces : *profilling*, ...). A ce jour plusieurs systèmes répondent au standard *MPI*. Cependant, comme nous l'avons cité auparavant, les solutions *LAM* et *MPICH* se distinguent réellement et prédominent dans l'unvivers des architectures parallèles. De plus, il est important de faire remarquer que plusieurs fonctionnalités du standard *MPI-2* ne sont pas encore incluses au sein de ces deux architectures (la plupart étant en cours de développement). En 1997, un projet intitulé *Interoperable MPI*[8] a débuté dans le but de définir l'ensemble des spécifications nécessaire pour interconnecter différentes implémentations du standard *MPI*. L'ensemble des critères et contraintes définis a été rassemblé au sein d'un rapport paru en juin 2000.

3.1.2 *LAM*

Développé depuis 1989, l'architecture logicielle de *LAM*[9] était relativement lourde, complexe et peu documentée. Récemment, une nouvelle implémentation [SL03] a été présentée. Basée sur les concepts de composant logicielle, cette version facilite l'intégration de techniques expérimentales au sein du modèle. C'est cette nouvelle architecture que nous présentons.

[5] Un unique programme (*SP*), exploitant les primitives d'échanges afin de partager de multiple-données (*MD*), est lancé en parallèle par l'intermédiaire de la commande `mpirun`. Il est, toutefois, possible «d'émuler» une programmation de type *MPMD* en s'appuyant sur les communications inter groupes proposées dans la seconde version du standard.

[6] Opération de type *scatter/gather*.

[7] Un tutoriel complet et rigoureux est proposé par le centre *IDRIS* à l'adresse suivante : `http://www.idris.fr/data/cours/parallel/mpi/choix\_doc.html`.

[8] `http://impi.nist.gov/`.

[9] *Local Area Multicomputer*, `http://www.lam-mpi.com`.

Le *SSI*[10] est constitué de deux principaux modules, interagissant avec le système d'exploitation (cf. figure 3.1) :

La couche *MPI* , elle se divise en deux sous-couches : la couche haute implémente les fonctions *MPI*, la couche basse est une multitude de composant qui implémente les spécifications*MPI* :

- Le *RPI*[11] gère les communications «point à point» ; il a été le premier composant redéveloppé sous une approche modulaire : l'intérêt étant de pouvoir supporter au sein d'une même installation *LAM / MPI* plusieurs supports d'échanges[12]. A ce jour, il est possible de dialoguer au dessus de *TCP*, au sein d'architecture à *mémoire partagée* ou encore via le réseau *Myrinet* (une version évoluée de *TCP* pour le module *CR* est également disponible).
- Le *Coll* intègre l'ensemble des algorithmes collectifs. Chaque module implémentant ce composant intègre deux versions de chaque opération collective : une pour les échanges intra-communicateur et une pour les actions inter-communicateurs. Ainsi, même si l'implémentation *MPI / LAM* ne fournit pas encore le code requis pour ce type d'opération, le concept de communication inter-communicateur a d'ores et déjà était pris en considération. De plus, la sélection d'un module peut être au choix du développeur : il est alors possible de définir un module par communicateur par exemple (`MPI_COMM_WORLD` emploiera un module `X` alors qu'un autre communicateur exploitera le module `Y`). Enfin, le développement d'un module de type *Coll* peut reposer ou non sur l'exploitation du composant *RPI*. En effet, si ce dernier fournit les routines de communications «point à point», l'architecture *SSI* n'impose pas une dépendance entre les modules, au contraire, elle permet une liberté dans la construction des modules. Ainsi, ils peuvent s'appuyer sur des composants «voisins», exploiter une structure hiérarchique (création d'arbre d'opérations collectives) ou encore être totalement distinct des sous-systèmes déjà existants.
- Le *CR* apporte de nouvelles routines offrant un contrôle des processus *MPI* en cas de défaillance (points de vérification, redémarrage de tâches, ...). Ce composant novateur[13] complète le standard *MPI 2* en apportant une plus grande fiabilité dans le déroulement des tâches *MPI*. D'un point de vue très générale, son interface est relativement simple, elle se compose des trois routines suivantes : `checkpoint` (sauvegarde l'état de la tâche), `continue` (donne suite à l'exécution après une vérification) et `restart` (redémarre un tâche après défaillance selon la dernière sauvegarde de son état). Pour que ce composant est un sens au sein de l'architecture logicielle *LAM / MPI*, il est impératif que les autres composants tiennent compte de ces différents états (chaque module doit pouvoir sauvegarder son état et être redémarrer par la suite) : le module *CRTCP* du composant *RPI* en est un exemple.

Chaque sous module possède la possibilité d'interagir directement avec le système d'exploitation (le *RTE* est court-circuité).

l'environnement d'exécution, *LAM RTE* consiste à l'ensemble des démons utilisateurs (*lamd*). La commande `lamboot` instancie un démon *lamd* sur chaque nœud participant à l'exécution. Ces démons fournissent les services de passage de message, de contrôle des processus ainsi que la gestion des entrées/sorties. la

[10] Le *System Service Interface* est le support d'éxécution de *LAM / MPI*.

[11] *Request Progression Interface*.

[12] Plusieurs modules implémentant le même composant peuvent coexister au sein d'un processus *MPI*, ce qui n'était pas possible avec les versions antérieures.

[13] De plus amples détails sur ce composant sont disponibles sur le site de *LAM / MPI*.

commande `lamhalt` permet d'arrêter le *RTE*. Le module *Boot* fournit les fonctionnalités nécessaire pour démarrer l'environnement d'exécution *LAM / MPI* ; il repose sur différents modules influencés par les commandes `rsh/ssh`. Des modules spécifiques du composant *Boot* permettent d'utiliser *LAM / MPI* par l'intermédiaire de différents environnements comme le gestionnaire de grappes *PBS* ou encore l'environnement *Globus*.

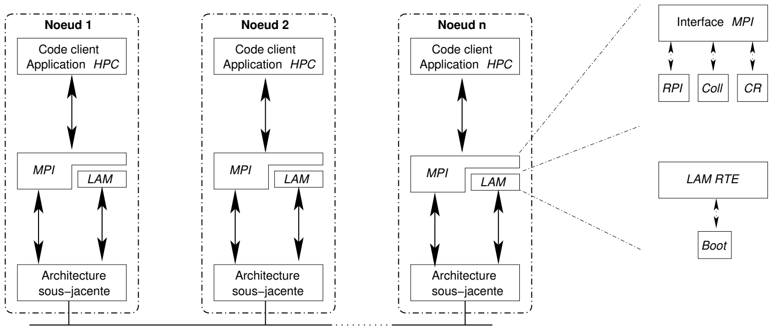

FIG. 3.1 – Architecture logicielle *LAM/MPI*

La possibilité d'exploiter de manière sélective la «brique» implémentant le service souhaité est un caractéristique clé de ce nouveau modèle. Ces «sous-modules» paramétrables lors de la compilation mais également lors de la phase d'exécution rendent la couche *LAM / MPI* moins lourde et plus approprié à l'application *HPC* elle-même (seules les fonctionnalités nécessaires sont chargées).

La dernière version de *LAM / MPI* (7.0.2) implémente la totalité des spécifications du standard *MPI 1* et la plupart des routines *MPI 2*. Les directions futures définies par l'équipe du laboratoire *OSL* de l'université de l'Indiana restent dirigées vers la restructuration du code en module. L'idée principale étant de proposer un module pour chaque aspect du standard *MPI* (contrôle de processus, topologie des tâches, module d'entrées/sorties[14], ...). Parallèlement, il est impératif que d'autres modules implémentant les composants soient mis en place (modules *RPI* pour les nouveaux supports réseaux ultra-rapides, passage à l'échelle, ...). Les dernières versions de *LAM / MPI* sont des projets «open-sources» sous licence BSD.

3.1.3 *MPICH*

Le projet *MPICH* est partie de la détermination de construire une implémentation simultanément à la conception des spécifications du standard *MPI*. Par cette contrainte volontairement imposée, il a été possible d'exposer rapidement les difficultés rencontrées par les développeurs vis à vis des spécifications «théoriques» suggérées. De plus, l'utilisation des sous-systèmes déjà existants comme *p4* [BL94], *Chameleon* [GS93] ou encore *ZipCode* [SSD$^+$94] a largement contribué au développement du modèle. Ainsi, il a été possible de fournir une version stable, portable et surtout performante de *MPICH* relativement peu de temps après la finalisation des spécifications de la première version de *MPI* (la solution est par conséquent devenue très rapidement un point de référence).

[14] Actuellement, les spécifications *MPI I/O* sont supportées par la solution *ROMIO*.

L'architecture logicielle de *MPICH*, [GLDS96], est constitué d'une superposition de couches (figure 3.2) pouvant être regroupées en deux principaux sous ensembles : le premier comprend l'ensemble des routines proposées par l'interface *MPI* alors que le second module, de plus bas niveaux, prend en charge toutes les interactions avec les différents composants matériels. Ces couches inférieurs souvent malencontreusement rattachées au sous-système connu sous le nom d'*Abstract Device Interface* (*ADI*) assurent la portabilité du modèle *MPI*. L'*ADI*, composant central de la bibliothèque *MPICH*, est le point de rattachement entre les pilotes souvent propriétaires (fournis par les industriels) et l'interface *MPI* employé par les applications *HPC*[15]. D'une manière générale, le développement d'une couche de l'*A.D.I.* est complexe et doit être minutieuse afin de tirer réellement partie des capacités offertes par les matériels sous-jacent. Il doit impérativement inclure les fonctionnalités suivantes : gérer l'envoi et la réception des messages (gestion des files d'attentes, délivrance d'acquittements aux couches hautes), déplacer et/ou copier les données entre les couches hautes de la librairie et les couches internes à l'*A.D.I.*, informer les couches hautes du comportement de l'application (nombre de processus, défaillance possible, ...).

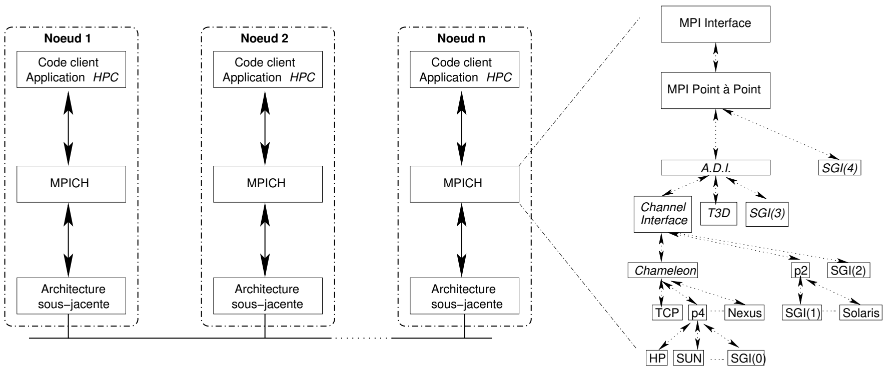

FIG. 3.2 – Architecture logicielle *MPICH*

Il est intéressant de noter que de plus en plus d'industriels proposent la totalité des couches inférieurs de l'*A.D.I.* au sein d'un unique module. Les modules de *SGI* illustrent tout à fait ces propos et montrent la flexibilité du modèle en proposant une «brique» à tous les niveaux. La mise en œuvre du module *SGI(4)* permettra à une application *MPI* d'exploiter la totalité des capacités physiques fournies par une architecture *SGI*.

Nous verrons que la structure en deux sous modules : les couches hautes et l'*A.D.I.* a également été reprise au sein du modèle *ROMIO* (cf paragraphe 3.3.5). Actuellement la dernière version stable de *MPICH* est la 1.2.5 (janvier 2003). Récemment, l'équipe d'Argonne vient de mettre à disposition une version *béta* de *MPICH 2*. Le but de cette nouvelle version est de fournir une implémentation des spécifications *MPI* prennant en compte les problèmes tel que le passage à l'échelle avec des architectures hétérogènes (*SMP*, grappe, ...). Elle devrait servir également comme support pour les nombreux

[15]Une couche interne, *Channel Interface*, propose les «briques» rudimentaires afin de rattacher la structure *MPICH* à un pilote spécifique

travaux de recherches autour de *MPI* et des nouvelles directions à prendre pour le développement de nouveaux environnements de programmation parallèle.

3.2 *MPI I/O*

Grâce au modèle comme *MPI*, l'écriture d'applications parallèle *HPC* est devenu chose courante. La difficulté rémanente est la gestion des entrées/sorties. Comme nous avons vu lors du second chapitre, un grand nombre de solutions ont été suggérées à la communauté durant ces dix dernières années. Néanmoins là encore, l'absence de standard et la large diversité des interfaces rendaient impossible le développement de programme «portable». C'est dans le cadre de l'extension de la norme *MPI* que des spécifications à l'encontre des entrées/sorties ont été établies puis regroupées sous l'appellation *MPI I/O*[16] [CFF+01],[TGL99b]. D'un point de vue sommaire, *MPI I/O* est une interface permettant de décrire les entrées/sorties parallèles au sein des applications *MPI*. L'idée fondamentale consiste à modéliser une entrée/sortie comme une opération d'échange de message : une écriture au sein d'un fichier peut être comparée à l'envoi d'une message alors qu'une lecture est semblable à une réception.

Partant de ce concept et en prenant en considération l'ensemble des remarques provenant des travaux précédemment réalisés (cf. chapitre 2), l'interface *MPI I/O* a été élaborée. il recouvre les problématiques tels que :
- le partitionnement de fichiers par rapport à une topologie de processeurs (mise en œuvre des «masques de fichiers» permettant d'appliquer les techniques de *Data Sieving* à l'échelle d'un nœud, cf. figure 3.3),
- l'accès en provenance de plusieurs entités, accès concurrents ou entre-lacés (en s'appuyant également sur les masques de fichiers afin de gérer les opérations collectives),
- l'asynchronisme dans le but de «maximiser» le recouvrement calcul / *I/O*.

Au sein de l'approche *MPI I/O* , les méthodes d'accès sont caractérisées par trois aspects : le positionnement (déplacement au sein du fichier, partage des descripteurs), le type d'appels (bloquant ou réalisé en arrière-plan) et enfin la coordination (opérations indépendantes / collectives). Chaque appel va engendré différentes optimisations permettant de diminuer ou de rendre transparent la latence *I/O*. Il ne m'a paru intéressant de détailler l'ensemble des spécifications *MPI I/O* au sein de ce rapport. Tout d'abord, comme il a été précédemment indiqué de nombreux documents s'attardant sur ce point sont disponibles sur l'Internet. De plus, une description minutieuse du standard n'aurait qu'alourdi la lecture sans pour autant apporter une réelle contribution. J'ai choisi de les regrouper au sein d'un tableau 3.2 selon les remarques et méthodes précédemment réalisées (cf chapitre 1). La caractéristique clé du modèle est la possibilité d'accéder à plusieurs portions de données non contiguës à partir d'un simple appel. La figure 3.4, emprunté d'une présentation de l'implémentation *MercutIO*, vient compléter le tableau en proposant une vue globale de la richesse de l'interface *MPI I/O* .

Remarque : Une manière simple de porter une application de l'interface *UNIX* standard à celle de *MPI I/O* est de remplacer tous les appels *UNIX* avec leur équivalent *MPI I/O* , [TGL98, TGL02]. Néanmoins, comme il est montré, même si l'approche

[16]Une introduction au modèle est disponible à l'adresse suivante :
`http://hpcf.nersc.gov/software/libs/io/mpiio.html`.

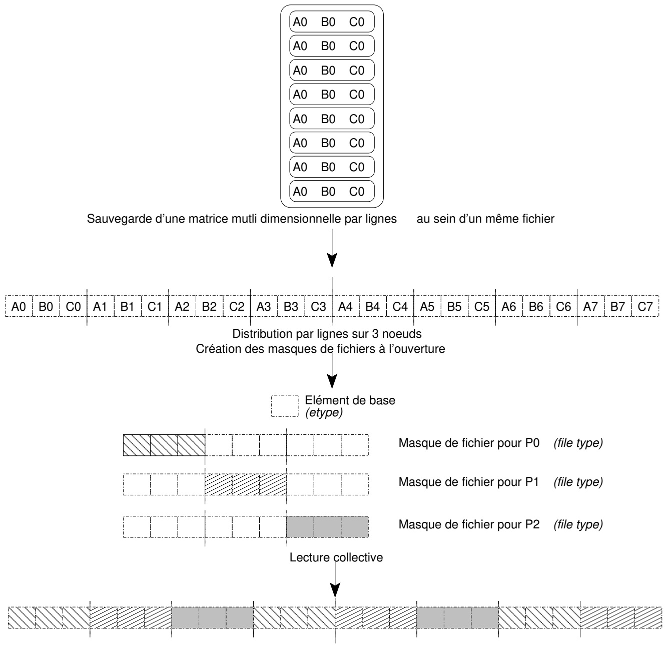

FIG. 3.3 – *MPI I/O* - Masque de fichiers
Les masques de fichiers, définis lors des opérations d'ouvertures, sont le moyen employé au sein du modèle *MPI I/O* afin d'apporter la connaissance requise pour la mise en œuvre de techniques comme le *Data Sieving* ou encore *Two Phase*. L'élément *élémentaire*, *etype* permet d'assurer une compatibilité et une intégrité entre le positionnement physique et logique.

simpliste de portage ne pénalise pas les performances, il est primordiale d'exploiter diverses caractéristiques de la bibliothèque *MPI* (et des couches *MPI I/O*) afin d'obtenir de réelles gains (Types dérivés *MPI*, masques de fichiers, opérations collectives, ...) en maximisant la taille de chaque requête.

Le modèle *MPI I/O* ne propose pas à l'instar de systèmes comme *Vesta*, *Galley*, ...(cf. chapitre 2) A de routines capables de gérer le placement physique des données sur disques. En effet, le consortium *MPI* n'a pas souhaité intégrer ce type de service : la correspondance entre vue logique et vue physique est dépendante des unités de stockages employées, par conséquent, fournir des routines de placement diminuerait considérablement le coefficient de portabilité du modèle *MPI I/O* (et donc *MPI*). Dans un souci de souplesse, *MPI I/O* a prévu un ensemble de routines «génériques» permettant de transférer diverses informations depuis l'application vers les couches basses du modèles. Il est ainsi possible pour une application de transférer une ensemble de paramètres afin d'optimiser les entrées/sorties au niveau matériel. Toutefois, ces routines ne sont régies par aucune règle *MPI I/O* spécifique ainsi rien n'impose leurs présences au sein des couches basses d'une implantation *MPI I/O* .

Parallèlement, il intéressant de remarquer que les notions comme la consistance ou bien l'atomicité des opérations d'entrée/sortie ne sont pas assurées par défaut au sein du modèle. Partant du constat qu'une application parallèle délivre rarement des accès concurrents recouvrants de type Lecture/Ecriture, il a été volontairement choisi de ne pas en tenir compte. Toutefois, le modèle propose un mode «protégé» capable

Positionnement	Mode	Coordination	
		Indépendant (*Data Sieving*)	Collective (*Two-Phase*)
Offset explicite	bloquant (synchrone)	`MPI_FILE_READ_AT`	`MPI_FILE_READ_AT_ALL`
		`MPI_FILE_WRITE_AT`	`MPI_FILE_WRITE_AT_ALL`
	non bloquant (asynchrone)	`MPI_FILE_IREAD_AT` `MPI_FILE_IWRITE_AT`	`MPI_FILE_READ_AT_ALL_BEGIN` `MPI_FILE_READ_AT_ALL_END` `MPI_FILE_WRITE_AT_ALL_BEGIN` `MPI_FILE_WRITE_AT_ALL_END`
Descripteur de fichier individuel	bloquant (synchrone)	`MPI_FILE_READ`	`MPI_FILE_READ_ALL`
		`MPI_FILE_WRITE`	`MPI_FILE_WRITE_ALL`
	non bloquant (asynchrone)	`MPI_FILE_IREAD` `MPI_FILE_IWRITE`	`MPI_FILE_READ_ALL_BEGIN` `MPI_FILE_READ_ALL_END` `MPI_FILE_WRITE_ALL_BEGIN` `MPI_FILE_WRITE_ALL_END`
Descripteur de fichier partagé	bloquant (synchrone)	`MPI_FILE_READ_SHARED`	`MPI_FILE_READ_ORDERED`
		`MPI_FILE_WRITE_SHARED`	`MPI_FILE_WRITE_ORDERED`
	non bloquant (asynchrone)	`MPI_FILE_IREAD_SHARED` `MPI_FILE_IWRITE_SHARED`	`MPI_FILE_READ_ORDERED_BEGIN` `MPI_FILE_READ_ORDERED_END` `MPI_FILE_WRITE_ORDERED_BEGIN` `MPI_FILE_WRITE_ORDERED_END`

TAB. 3.1 – Routines d'accès *MPI I/O*

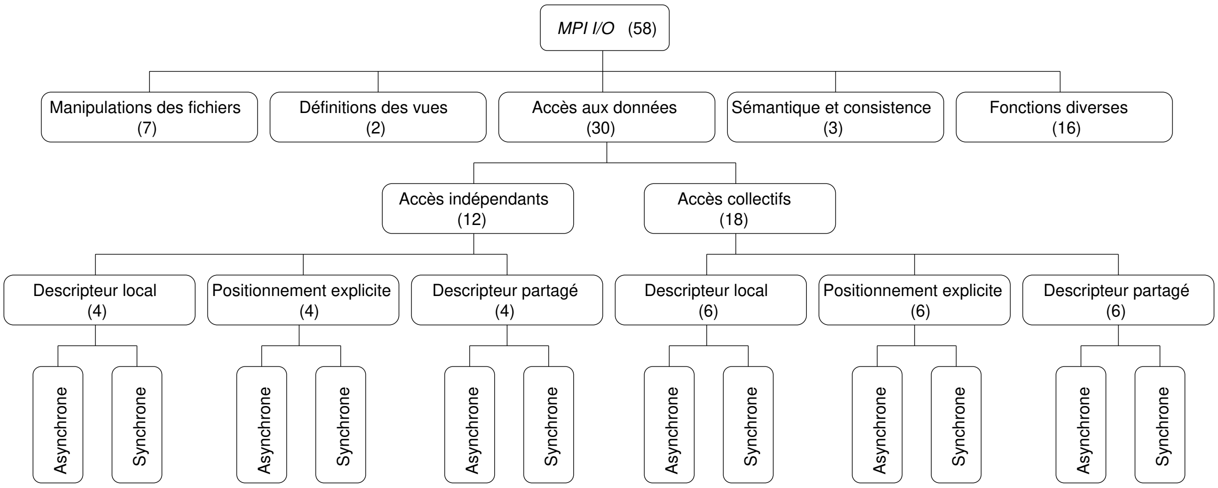

FIG. 3.4 – Organisation de l'ensemble de l'interface *MPI I/O*

d'assurer l'intégrité des données au sein d'une application *MPI* (*MPI_File_sync* pour la consistance et *MPI_File_Control* afin de garantir l'atomicité).

3.3 Implémentation *MPI I/O*

De multiples implémentations [FWNK96], [TGL99b], [GH01], [SS00], [JMM+96], [PTH+01] ...ont été proposées depuis la présentation du standard. Nous allons décrire au sein de cette partie les plus connues.

3.3.1 *PMPIO*

Développé par les centres de recherches de la *NASA*, *PMPIO*, [FWNK96], a été présenté en 1996 dans le but de «valider» les diverses étapes de l'élaboration des spécifications *MPI I/O* . Le principal but était d'apporter une contribution dans la définition des spécifications *MPI I/O* par l'intermédiaire d'une implémentation expérimentale permettant de confronter le modèle théorique au développement pratique. L'idée étant de pouvoir proposer à l'ensemble de la communauté scientifique une implantation susceptible de servir par la suite de référence (comme point de démarrage) pour tout autre développement du modèle.

Lors de la conception de *PMPIO*, la contrainte de «portabilité» a été fortement éprouvée en offrant le support pour un grand nombre d'architectures (*IBM-SP2*, *SGI*, *cluster Sun*, *Intel Paragon*, ...). Parallèlement, l'implémentation de la bibliothèque a permis d'étudier et d'analyser la modularité au sein du modèle *MPI I/O* . L'intention consistait à décomposer le modèle afin de retrouver les différentes «briques» fondamentales du système et expérimenter divers techniques d'optimisations (*Collective Buffering*, *Data sievieng*, approche client/serveur ou bibliothèque, ...).

Afin d'analyser les performances du standard *MPI I/O*, *PMPIO* a intégré les routines d'accès indépendantes et collectives, la gestion des descripteurs de fichiers non-partagés et l'ensemble des types dérivées *MPI*[17]. La mise en place de la technique de *Collective buffering* (cf. chapitre1.2.3) a permis d'obtenir des gains jusqu'à vingt fois supérieurs aux routines d'entrées/sorties *Fortran*. Toutefois, le modèle *PMPIO* a notamment remontré que les méthodes tendant à agréger les requêtes afin de maximiser la bande passante *I/O* ne fournissent pas de réels gains sur les architectures de type *SMP* possédant un espace mémoire conséquent (les gestionnaires de caches «faussent» totalement les résultats lors des opérations d'écritures par exemple).

3.3.2 *MPI I/O* au dessus de HPSS

En 1996, dans le cadre projet *SIOF* (*Scalable Input/Output Facility*[18]), une implémentation des couches *MPI I/O* au dessus du système de fichiers distribué *HPSS* (*High Performance Storage System*[19] [WC95, HGFW02]) a été présentée, [JMM+96]. Cette implémentation de *MPI I/O* , proposée au sein des travaux *SIOF* a été conçu avec les contraintes suivantes :

- l'utilisation efficace des ressources *I/O* avec une répartition équitable de l'ensemble des charges,
- la suppression de toutes méthodes ou approches susceptibles d'engendrer des goulets d'étranglements,
- et enfin la minimisation des points de synchronisation entre les processus de l'application.

[17]Les pointeurs partagés, les opérations asynchrones ainsi que la gestion des erreurs ne figuraient pas au sein du modèle expérimental.

[18]Le but du projet *SIOF* est la mise en place d'un système de stockage tenant en compte les problématiques des entrées/sorties parallèles et de passage à l'échelle : `http://www.llnl.gov/liv\_comp/siof/siof.html` .

[19]*HPSS* est un système de fichiers hiérarchique conçu pour supporter des architectures conséquentes (composées d'un très grand nombre de nœuds). Il se compose de plusieurs couches offrant divers niveaux de sécurité pour l'accès aux informations.

D'un point de vue logicielle, le modèle se présente de la manière suivante (cf. figure 3.5) ; sur chaque processeur, deux processus légers sont instanciés : le *thread* «client» contient le code de l'application ainsi que l'implémentation de l'interface *MPI I/O* alors que le second, le *thread* «serveur» est employé pour interagir avec le système *HPSS*. Chaque tâche «serveur» est responsable d'une partie de l'arborescence du système de fichier ; cette répartition évite de centraliser les requêtes en un unique point (susceptible de goulet d'étranglement et de défaillance majeure). Cette technique, plus connu sous le nom de modèle de serveur distribué, a largement était repris dans de nombreux systèmes (*GFS*, *GPFS*,...).

D'un point générale, lorsqu'un client requiert l'exécution d'une opération *I/O* , il transmet un message *MPI* au *thread* «serveur» sur le nœud approprié. Après avoir récupéré les «méta-informations» auprès du contrôleur *HPSS*[20], le serveur va alors créer un *thread* «driver» ; ce dernier gérera la totalité du transfert *HPSS* et retournera un message «résultat» directement au client. Dès que le client reçoit cette réponse, l'appel *MPI I/O* est terminé et l'application continue son exécution.

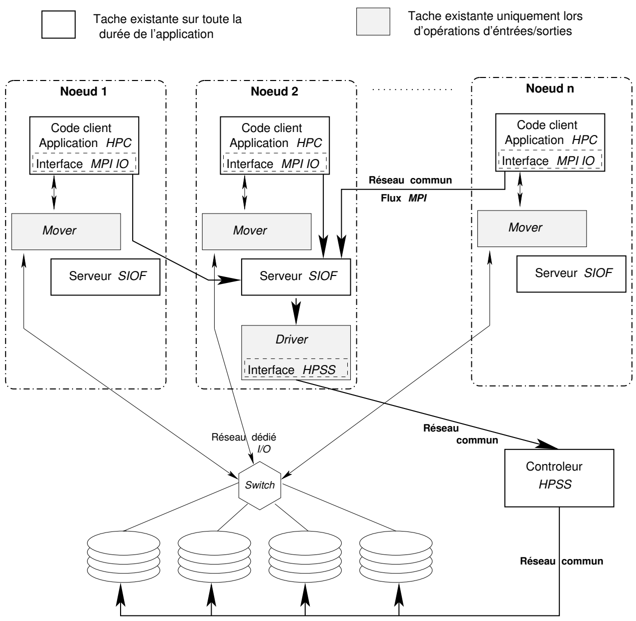

FIG. 3.5 – *MPI I/O* et *HPSS*
SIOF profite de l'architecture liée à *HPSS* afin de réaliser les opérations de transferts de données directement depuis les clients vers unités de stockages. L'exploitation de *threads* permet une parallélisation des tâches à effectuer lors d'une opération d'entrée/sortie parallèle. La situation illustrée ci-dessus présente les différentes entités mises en œuvre lors d'une opération collective.

Nous allons nous attarder plus en détails sur chaque groupe d'opérations :

[20]Le terme de contrôleur est employé ici afin de simplifier la compréhension du déroulement des opérations. Toutefois cette entité correspond au serveur de méta-informations de l'architecture *HPSS*.

Ouverture/Fermeture, l'ouverture d'un fichier étant définie comme une opération collective, chaque client délivre une requête à destination du même serveur (une fonction de hachage permet de retrouver le serveur responsable du fichier) : dès la réception d'un des messages, le serveur réalise les opérations nécessaires sur les méta-informations (en y accédant directement au sein d'un cache local ou en les récupérant préalablement auprès du contrôleur *HPPS*). Cette technique permet d'optimiser les flux réseaux, en effet, pour n requête(s) «cliente(s)», un unique message au plus est dirigé vers le contrôleur *HPSS*. De plus, il n'est pas utile de synchroniser les processus clients sur une ouverture de fichier. A Contrario, la fermeture d'un fichier, qui s'avère également être une opération collective, impose une synchronisation : il est impératif d'attendre la réception de tous les messages de fermetures avant de transmettre concrètement le message de fermeture au contrôleur *HPSS* (le fichier ne doit en aucun cas être fermé si certains nœuds sont encore entrain de réaliser divers accès).

Structure des données et masque de fichier, Les couches internes de la partie cliente *MPI I/O* transforme les types *MPI* en un format interne à l'architecture et permettant de construire les descripteurs requis par les système *HPSS*. De ce fait, la solution *SIOF* permet de regrouper les accès au sein d'un même processeur. La «centralisation» des messages vers un *thread* serveur spécifique permet également d'aggrèger les requêtes en direction d'une même unité de stockage.

Lecture/Ecriture indépendante, lors de telles opérations, le *thread* client commence par créer une tâche, «mover» qui va copier les données depuis le tampon temporaire de l'application cliente B vers l'unité de stockage. Une fois que le «*mover*» a été instancié, le client transmet, un message *MPI* au serveur concerné. Ce dernier va une nouvelle fois lancer une nouvelle tâche «driver» afin d'interagir avec les services *HPSS* concernés. Ces derniers prendront en charge le transfert des données. Comme pour les opérations précédentes, le *thread* «driver» retourne le résultat directement au client. L'opération *MPI I/O* se termine dès réception de ce résultat.

Lecture/Ecriture collective, chaque client émet un message en direction du serveur déterminé. Le serveur construit alors une «liste d'accès» en ajoutant à chaque nouvelle réception de message la portion de données exigée. l'approche *SIOF* se distingue des méthodes de type *Collective Buffering* que nous avons décrites jusqu'ici. En effet, elle intègre un nouveau paramètre permettant de déclencher le transfert effectif des données à un moment prédéterminé. L'utilisation d'une variable seuil va causer la mise en œuvre du transfert des données entre les *threads* «mover» et les unités de stockages. Ainsi, il est possible que la requête *HPSS* transmise au contrôleur par la tâche «driver» soit envoyée avant que la totalité des messages composant l'opération collective n'ait été reçue par le serveur. Pour éviter tous conflits, le serveur maintient à jour le nombre de nœuds qui ont été pris en considérations. La détermination du seuil de déclenchement joue un rôle capitale dans le gain pouvant être obtenu. A ce jour, l'opération de transfert ne débute que lorsque la liste des accès est complète (l'ensemble des portions de données souhaitées par l'opération collective y apparaît).

Opération asynchrone, chaque serveur exploitant des *threads* «driver» a la possibilité de réaliser des opérations de manière asynchrone. Toutefois les requêtes à destination d'une même ressource sont sérialisées afin de préserver les contraintes d'atomicité au sein du modèle.

Lors d'une premier constat, l'équipe de *SIOF* a défini deux optimisations majeures à apporter au modèle. Tout d'abord, la distribution (ou répartition) équitable de l'arborescence entre les différents *threads* serveurs impose une étape de communication (dans le meilleur des cas locale à un nœud) pour toutes opérations d'entrées/sorties. Il serait intéressant de pouvoir éviter ce surcoût lors d'accès indépendants ; en effet, si la centralisation des requêtes lors d'opérations collectives apporte un réel plus dans le choix de la stratégie, elle n'est d'aucune utilité dans les autres cas. Par ailleurs, il soulève l'intérêt potentiel des primitives de type `MPI_Reduce`[21] dans le codage des routines collectives afin d'exploiter au mieux le trafic réseau. Néanmoins, l'usage de telles méthodes implique obligatoirement des points de synchronisations entre les nœuds. Une étude de ces deux approches permettrait de voir dans quelle cas l'une ou l'autre sont optimales.

[HJMY00] complète la description du modèle en s'attardant sur la fonctionnalité de transfert (*third-party transfer*[22]) proposée par le système *HPSS* et exploitée au sein du modèle *SIOF*. Il compare cette technique à la méthode *Disk-Directed I/O* (cf. paragraphe 1.2.3) : en effet, lorsqu'un fichier est réparti sur plusieurs unités de sauvegarde et que plusieurs nœuds accèdent à différentes portions, le transfert des données depuis les unités de stockage est exécuté en parallèle directement vers les machines concernées (il n'y a pas de phases de permutation des données ou de recopies sur des entités distantes). Il soulève enfin la difficulté de réaliser une correspondance entre vues *MPI* (par l'utilisation des types dérivés) et vues *HPSS* (descripteur de trame de transfert) dans le but de maximiser les performances susceptibles d'être délivrer par l'architecture *HPSS*.

3.3.3 *MPI I/O* au dessus de GPFS[23]

Le développement d'une version optimisée de *MPI I/O* pour le système *GPFS* a été présenté en fin 2001. Ce projet, [PTH$^+$00, PTH$^+$01], entre dans le cadre des divers travaux menés par *IBM* dans la problématique des entrées/sorties[24]. Le but est de pouvoir exploiter au maximum les fonctionnalités offertes par la nouvelle interface de *GPFS 1.3.* .

GPFS a introduit le concept de *Data Shipping* dans le but de gérer de manière optimale les accès concurrents. Chaque bloc de données *GPFS*[25] est rattaché à un agent ; cette entité va alors se comporter comme un serveur pour l'ensemble des requêtes adressées à cette portion (en d'autres termes, seules les agents réalisent concrètement des opérations d'entrées/sorties sur les unités de stockages). La première optimisation suggérée au sein de cette implantation de *MPI I/O* est de réaliser une distribution des blocs *GPFS* sur l'ensemble de l'architecture en, premièrement, déployant un agent *I/O* par nœud *MPI* et deuxièmement, en réalisant la distribution de manière à obtenir une correspondance optimale entre le découpage physique des blocs *GPFS* et la vue logique (cf figure 3.6). Ce procédé diminue considérablement le nombre de transfert en terme de messages et en terme de données entre les multiples agents *I/O* d'une part

[21] Les primitives «*reduce*.

[22] Le transfert de données entre deux processeurs peut être réalisé par un troisième processeur indépendant.

[23] *General Parallel File System*, une présentation concise du système apparaît dans [Leb02].

[24] Une implémentation avait précédemment été réalisé au dessus du système de fichiers *PIOFS* (cf. chapitre 2.2.1).

[25] Nous rappelons qu'une des caractéristiques du modèle *GPFS* est la grandeur de ses «blocs systèmes» (en moyenne 256Ko, mais cette taille peut varier de 16Ko à 1Mo).

mais également permet d'éviter les éventuelles conflits de type écriture concurrentes et la nécessité de faire intervenir le *token manager* spécifique à *GPFS*.

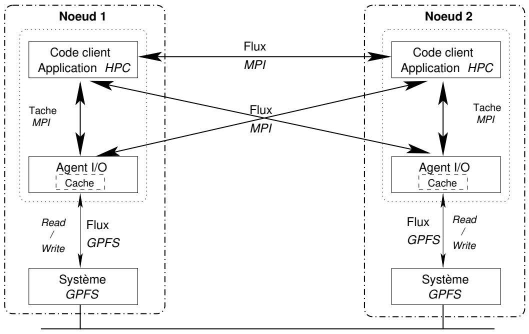

FIG. 3.6 – *MPI I/O* et *GPFS*
IBM a mis au point une implémentation des couches *MPI I/O* directement en dessous des couches *MPI* permettant d'exploiter de manière optimale les fonctionnalités offertes par le système de fichiers *GPFS*. Chaque agent *I/O* est responsable d'une portion de données. Dans un premier temps la distribution était réalisée de manière cyclique ; par la suite, une répartition plus réfléchi a été intégrée, elle tient compte de la correspondance vue logique et vue physique.

Un gestionnaire de cache inclus au sein des agents d'entrées/sorties exploite les «masques» de fichiers afin de d'exécuter du pré-chargement efficace (*prefetch*, cf. paragraphe 1.4). Lorsqu'un agent *I/O* observe que l'opération qui va être réalisée est régie par un patron défini (cf. paragraphe 1.1.5), il «laisse» *GPFS* effectuer ses propres techniques de pré-chargements déjà éprouvées. Lorsque les opérations interviennent de manière plus ou moins aléatoire, l'agent construit une liste d'intervalles qu'il récupérera de manière anticipée par la suite (une méthode comparable à la technique de *Data Sieving* est effectuée sur chaque bloc *GPFS*, la liste représente l'ensemble des portions «recouvrantes» sur la totalité des blocs *GPFS* concernés par l'opération).

Des améliorations ont été également apportées aux niveaux des opérations collectives : un mode «activable» permet de réaliser une première phase qui va déterminer les agents *I/O* réellement concernés. Grâce à cette technique, il est possible de minimiser le nombre de messages et les points de synchronisations requis pour l'opération. Dans le cas où l'opération collective nécessite l'implication de la totalité des agents *I/O* , l'étape permettant de définir le meilleur arbre de transfert est inutile. Le temps requis par cette phase augmente le temps de réactivité global de l'opération. Il est, par conséquent, important de bien connaître le «patron» des données afin d'employer de manière bénéfique cette optimisation. Une seconde optimisation a été proposée par l'équipe d'*IBM*. En effet, il suggère l'utilisation d'un double tampon dans le but de «paralléliser» ou de recouvrir les étapes qui constituent une opération collective[26]. Une opération collective se décompose en une phase de communication entre les tâches *MPI* et les agents *I/O* et un phase d'accès «concrets» au système *GPFS*. La mise en œuvre d'un second tampon temporaire rend possible le recouvrement des étapes lorsque celles-ci sont exécutées plusieurs fois d'affilée.

[26]Nous avons indiqué précédemment (cf . paragraphe 1.2.3) qu'il est tout à fait possible qu'une opération collective nécessite plusieurs itérations de la totalité de l'algorithme suites aux limitations des tailles des tampons temporaires.

Dans la majeure partie des cas, la mise en application des méthodes d'optimisations décrites auparavant est transparente, le modèle *MPI I/O GPFS* déduit de manière indépendante la politique à appliquer. Toutefois, un contrôle fin des différentes techniques impose l'utilisation de routines de «transfert d'informations» depuis les couches hautes vers les couches basses du modèle *MPI I/O* . Il est fondamentale de rappeler que l'exploitation de telles routines diminue considérablement la portabilité de l'application *HPC* . Les appels explicites au sein du code à la méthode `MPI_FILE_SET_INFO` ou le passage de paramètres spécifiques lors des primitives d'ouvertures rendent l'application dépendante au modèle *GPFS*. Un développement rigoureux imposerait de tester l'existence des variables et les modes spécifiques au modèle *MPI I/O GPFS*.

3.3.4 *MercutIO*

MercutIO, [Raj02], est une implémentation des spécifications *MPI I/O*. Parue en 2002, cette bibliothèque suggère un modèle totalement asynchrone : la totalité des opérations est réalisée par des routines non-bloquantes (les appels *MPI I/O* bloquants sont émulés au dessus des appels asynchrones). L'intérêt d'un modèle asynchrone est d'exploiter au maximum les phénomènes de recouvrement calcul/*I/O* .

Le modèle se scinde en deux parties :

La couche *BAFS*, est la couche basse de l'implémentation. elle permet d'interfacer les différents systèmes de fichiers sous-jacents Tirant partie des concepts objets, la couche est relativement adaptable à toutes nouveaux types d'architectures. Comparable à l'*ADIO*, elle propose une interface d'entrées/sorties point à point efficace entre les différents nœuds et les divers systèmes de stockages. *BAFS* permet en autre d'abstraire plusieurs systèmes de fichiers au même moment (utilisation du concept du système de fichier virtuel) ou d'exploiter de front des unités de sauvegarde via des librairies *I/O* à accès directe. L'asynchronie totalement intégrée au modèle et implantée au sein de ce sous-système : elle est mise en œuvre soit par l'intermédiaire de routines non-bloquantes fournies par les systèmes de fichiers sous-jacents soit par l'exploitation de *threads* dans le cas contraire. Par ailleurs, l'agrégat des requêtes au sein d'un nœud (*Data Sieving*) est également effectuée au sein du module *BAFS*. Ayant quantifier la notion de «degré de continuité» (cf. paragraphe 1.2.4) durant le phase de conception, le modèle *MercutIO* tient compte de la distance qui séparent chaque «bloc» de données contigus ; si celle-ci est inférieure à la taille la plus petite des portions alors l'approche *Data sieving* est mise en œuvre, dans le cas contraire, l'approche standard *UNIX* est employé.

L'interface *MPI I/O* , est implémentée par une couche haute. Elles intègrent la totalité des routines définies par les spécifications *MPI I/O* :
– pointeurs partagés, par l'utilisation des primitives de verrous proposées par UNIX (*file lock*),
– opérations collectives, mise en place de l'algorithme *Two-Phase*,
– vues logiques, ...
Elle assure également la cohérence des données entre les différents processus ainsi que l'atomicité des opérations.

A ce jour, *MercutIO* a été incorporé au sein d'une implantation récente du standard

MPI 2. Projet industriel à but commercial présenté en avril 2003, *ChaMPIon/Pro*[27] est disponible sous l'environnement *Linux* (*RedHat*).

3.3.5 *ROMIO*

Dans le cadre des travaux initiés par le laboratoire Argonne autour du standard MPI (cf. 3.1.3) et des entrées/sorties parallèles (cf. 2.4), une implémentation des spécifications *MPI I/O* , intitulée *ROMIO*[28], a été présentée en octobre 1997. Même si la structure hiérarchique (cf. figure 3.7) mise en œuvre est comparable à l'implantation *MPICH* exploitant l'*A.D.I.*, *ROMIO* a été conçu indépendamment et peu par conséquent être intégrée à différentes plate-formes *MPI*[29].

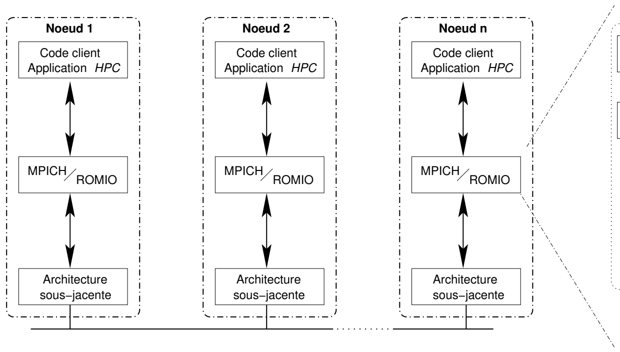

FIG. 3.7 – *MPI I/O* et *GPFS*

L'architecture *ROMIO* se scinde en deux parties : la première implémente les routines *MPI I/O* et est commune aux différents sous-systèmes composant la couche basse du modèle assurant la portabilité : l'*ADIO*. De nombreux travaux ont permis de faire évoluer le modèle, nous allons dresser dans une première sous partie partie, un état de la solution actuelle. Par la suite, nous nous attarderons sur les avancés spécifiques qui ont été réalisés afin d'optimiser les interactions avec le système de fichiers *PVFS*.

L'interface MPI I/O dans ROMIO

Comme nous l'avons indiqué, le but de ce rapport est de dresser un état du standard *MPI I/O* ; étudier son efficacité ainsi que les difficultés d'implanter ses spécifications au sein d'une solution concrète. L'implémentation *ROMIO* étant la plus déployée, il nous a paru logique de nous y consacrer de manière méthodique et rigoureuse [TGL98, TGL99b, TGL02, TGL99a, **?**]. Le bilan obtenu est présenté au sein de ce paragraphe. Nous y décrirons les principales routines et spécifications de cette solution en nous attardant sur les deux principales techniques implémentées afin d'optimiser les accès sur des données diffuses, à savoir le *Data Sieving* à l'échelle d'un nœud et l'approche *Two Phase* mise en place lors d'accès collectifs.

[27]`http://www.mpi-softtech.com`, Une version parallèle de *MPI* (*MPI / Pro*) est également développé au sein du même organisme.
[28]`http://www.mcs.anl.gov/romio.`
[29]A ce jour, *MPICH*, *HP MPI*, *SGI MPI*, *LAM MPI*, *NEC MPI*.

Ouverture et Fermeture de fichier : L'ouverture d'un fichier, `MPI_File_open`, est une opération collective. La plupart du temps, elle est émulée via un appel, au sein de chaque processus interne au communicateur, à la fonction UNIX `open`. Cependant, certains systèmes comme *Intel PFS* comporte une méthode `gopen`[30]. Dans de tel cas, il est plus favorable de faire appel à de cette fonctionnalité spécifique.
La fermeture, `MPI_File_close` consiste à appeler la méthode UNIX `close` sur chaque processus participant. Il est important de noter que l'appel à cette méthode nécessite la synchronisation de tous les processus qui ont réalisés l'ouverture préalablement.

Taille des fichiers : La taille maximale d'un fichier est déterminé par le type *MPI* : `MPI_offset`. Ce type utilisé est généralement codé sur 4 octets (2Go) ; toutefois, il est possible de redéfinir ce dernier afin qu'il soit codé sur une plage de 8 octets et par conséquent exploiter des fichiers de plus grande taille (proche des 10 «exa-octets»[31]) *ROMIO* intègre cette politique lorsque les systèmes de fichiers sous-jacent comprennent la gestion des fichiers supérieur à 2Go.

Déplacement : Les routines de déplacement de type `seek` dépendent du mode sélectionné lors de l'ouverture du fichier. Comme nous l'avons indiqué, *MPI I/O* offre la possibilité d'exploiter un descripteur commun (ou partagé) pour l'ensemble des processus commun à un même communicateur. Peu de systèmes de fichiers proposent une fonctionnalité similaire (descripteur partagé vs descripteur indépendant) ; la solution *ROMIO* doit par conséquent l'implémenter. Plusieurs méthodes coexistent afin de parvenir à la gestion de cette ressource critique. La première consiste à stocker cette information au sein d'un fichier et d'y accéder par l'intermédiaire d'un verrou. D'autres approches exploitent le potentiel fourni par les architectures où elles sont déployés : soit en exploitant les mémoires partagées, soit en utilisant un *thread* réalisant les opérations nécessaires au maintien des différents descripteurs. c'est la première technique qui a été incorporée au sein de *ROMIO* afin de maximiser la portabilité du système : un fichier contenant la valeur courante du descripteur est créé au sein du même répertoire où se trouve la ressource lors du premier accès et est éliminer lors de la fermeture. Lorsqu'un processus souhaite accéder au fichier, il prend un verrou sur le descripteur, le modifie, relâche le verrou et réalise son opération d'entrée/sortie.

Accès standard lecture/écriture Les accès contigus sont redirigés vers les fonctions `read` et `write` de chaque système de fichiers. Les fonctions plus performantes proposées par certains systèmes de fichiers sont exploitées par la couche correspondante de l'*ADIO* (`pread64`/`pwrite64` sur *XFS* par exemple).

Accès diffus, *Data Sieving* et *Two phases approach* : La mise à disposition de routines efficaces pour les accès non contigus a été un caractère prépondérant lors de la définition du standard *MPI I/O* . A l'instar des fonctions `readv` / `writev` disponibles sous UNIX mais qui elles sont limitées a six blocs distants, les primitives fournies dans *ROMIO* permettent de réaliser des opérations sur des données distantes aussi bien en mémoire via les types de données structurées *MPI* que sur les unités de stockages via l'utilisation des «*file views*», [TGL98]. Quatre groupes (ou mode d'accès) ont été ainsi établies (cf tableau **??**). Il est important de noter que les primitives `lio_listio` proposées au sein du standard *POSIX* offrent également la possibilité de réaliser en un unique appel de multiples

[30]«global open».

[31]*kilo,mega,giga,tera,peta et exa.*

Placement mémoire	Placement disque
Contigu	Contigu
Diffus	Contigu
Contigu	Diffus
Diffus	Contigu

TAB. 3.2 – «Patrons» (ou mode d'accès) au sein de *ROMIO*

requêtes. Cependant, chaque requête est exécutée de manière indépendante sans tenir compte d'éventuelles recouvrement ou tout autre optimisations susceptibles d'améliorer les performances.

Implémenté au sein de l'*ADIO*, *ROMIO* intègre deux politiques permettant d'optimiser ce type d'accès, [TGL99a] : la méthode *Data Sieving*(cf. 3.3.5) et l'approche collective *Two Phases* (cf. **??**).

Algorithme 1 La méthode *Data Sieving* dans *ROMIO*

Opération de lecture

tant que Requête non satisfaite **faire**
  Récupération des données sur disque au sein du tampon temporaire *ROMIO*
  {Rappel : la taille du tampon est limitée.}
  Extraction des données utiles depuis tampon temporaire vers l'espace réservé de l'application
fin tant que

Opération d'écriture

tant que Requête non satisfaite **faire**
  Récupération au sein du tampon temporaire de la portion recouvrante à partir du disque
  {Opération «*read-write-modified*»}
  Récupération et affectation des données en provenance de l'application au sein du tampon temporaire
  Ecriture de la portion recouvrante depuis tampon temporaire vers l'espace de stockage
fin tant que

Dans le cadre de l'agrégation au niveau d'un nœud (*Data Sieving*), la taille du tampon interne permettant le rapatriement des données est prépondérante. En effet, elle correspond à la taille maximale pouvant être recouverte. Dans le cas où les données sont diffuses sur une plus large étendue, l'opération de *Data Sieving* est alors réalisé en plusieurs itérations. Par ailleurs, nous insistons sur le fait que les "espaces" entre chaque bloc utile prennent par au coût total de l'opération (cf. paragraphe 1.2.1). Ainsi pour certains cas, la mise en œuvre d'une telle méthode peut s'avérer néfaste et une approche plus juste consisterait à activer ou non l'agrégation des requêtes d'entrée/sortie lors de chaque opération. En 1999, l'équipe d'Argonne a prévu d'intégrer cette amélioration au sein de l'architecture. Dans le cadre des opérations de type écriture, la portion accédée doit être verrouillée afin d'assurer l'intégrité des données (opération dite de «*read-modify-write*») : Un verrou est initialisé afin de protéger l'ensemble des données contenues dans la portion recouvrante qui est alors récupérée au sein du tampon temporaire ; les données conséquentes à l'opération sont modifiés puis le tampon est renvoyé vers l'unité de stockage avant de relâcher le verrou. Il est donc impératif que les systèmes de fichiers sous-jacents fournissent un mécanisme de «*lock*» afin de pouvoir proposer cette fonctionnalité[32].

Considérations techniques : le tampon temporaire a une taille de 4Mo par défaut pour les opérations de lecture et de 512Ko lors des écritures. les tailles diffèrent afin d'éviter les problèmes de contentions («*dead lock*») pouvant subvenir lors des opérations de type «*read-modify-write*. Toutefois, l'utilisateur a la possibilité de faire varier cette taille lors de l'exécution en exploitant les primitives de type «*hint*».

Algorithme 2 Approche *Two Phases* au sein de *ROMIO*

Opération de lecture

Création de la liste correspondant aux différents accès
{Une cellule de la liste est composée d'un <<offset» de départ et d'une quantité}

Transmission des bornes «min» et «max» de la plage recouvrante aux processus participants à l'opération collective

si Intérêt de mettre en œuvre l'opération d'agglomération collective == Vrai **alors**

  Répartition sur l'ensemble des processeurs sélectionnés : attribution des «*file domain*»
  {Possibilité de spécifier quels processus vont concrètement accédées aux disques}

  Mise en place de la politique d'échange liée à l'approche *Two Phases*
  {Echange de messages permettant à chaque processus de connaître la destination des données comprises dans son «*file domain*».}

  Détermination de la variable $max(ntime)$: échange collectif de la variable $ntime$
  {`ntime` : nombre d'itérations requises pour un processeur afin de récupérer toutes les données (taille du *file domain*/ taille du tampon de récupération)}

  $indice \leftarrow 0$
  tant que $indice < max(ntime)$ **faire**
    si $indice < ntime$ **alors**
      Récupération des données sur disque au sein du tampon temporaire *ROMIO* du processus
      {Rappel : la taille du tampon est limitée.}
    fin si
    Récupération et/ou transmission des données vers les processus adéquats
    {Utilisation des primitives non bloquantes *MPI* `send/receive`}
  fin tant que
fin si

L'algorithme présenté ci-dessus est réalisé sur chaque processus participant à l'opération collective. Dans le cadre d'une opération d'écriture seules les «deux phases» de l'itération sont inversées, toute la période d'initialisation de la boucle reste inchangée.

Dans un souci de compréhension des enjeux et des facteurs qui ont une influence sur les performances fournit par cette méthode nous allons nous attarder sur plusieurs points :

Phase d'initialisation : comme nous l'avons à maintes reprises indiqué, il est primordiale d'analyser préalablement le gain pouvant être obtenu. Une étude exhaustive sur l'ensemble des accès diffus s'avèrerait trop coûteuse au sein de la primitive collective. Les développeurs de *ROMIO* ont choisi d'intégrer

[32] *PVFS* ne permet par exemple pas de mettre en œuvre l'approche du *Data Sieving*.

un test simple permettant de valider ou non la mise en œuvre de l'agrégation collective : pour deux processus de rang *MPI* consécutif, l'expression suivant est vérifiée : $offsetBorneMax - P0 > offsetBorneMin - P0$. Dans le cas ou l'expression est fausse, la méthode *Data sieving* est proposée, l'opération collective prend alors fin.

Attribution des *file domains* : La répartition des accès peut être spécifiée préalablement dans le but de tenir compte de l'architecture matérielle sous-jacente ; des travaux sur le système *PVFS*, favorise par exemple la localité des données. Le but de cette affectation «plus juste» étant d'assigner les domaines de fichiers afin d'assurer un équilibrage de la charge de manière à limiter les transferts et minimiser le coût des communications par la suite.

Phase de récupération et/ou de distribution des données : Une étude approfondie de la méthode, [DT01] a été réalisée afin de déterminer l'importance de l'ordre des appels ainsi que celle de l'exploitation des routines asynchrones. Cette analyse a permis de définir une implémentation optimale de la phase de récupération/distribution des données : L'ensemble des requêtes non bloquantes de réception sont postées préalablement aux requêtes d'émission synchrones. L'implantation *ROMIO* respecte l'ordre défini par cette étude mais fait uniquement usage de primitives asynchrones. Le coût de l'approche collective est corrélée par l'utilisation de primitives d'échanges et de recopies, à l'implantation *MPI*. Au sein des *MPICH*, le masque des données est exploité afin de limiter les copies intempestives : si les données sont contiguës, la routine de réception exécute la copie directement au sein de l'espace mémoire de l'application ; dans le cas contraire, un tampon intermédiaire est réquisitionné afin de «replacer» les informations. Ce tampon peut être rempli en de multiples itérations de la boucle *Two phases* et éviter ainsi diverses copies vers les tampons de l'application.

Opération d'écriture collective : A la différence du *Data sieving*, les écritures agrégées de manière collective ne nécessitent pas d'opérations de type *read-modify-write*. En effet, la sémantique *MPI I/O* garanti une consistance basée sur les communicateurs. L'attribution des *file domains* assurent qu'aucun autre processus impliqué dans l'opération collective (et donc interne au même communicateur) n'accède à cette portion. Il n'est donc pas nécessaire de se préoccuper du comportement des processus extérieurs au communicateur (dans de tel cas où une consistance plus rigoureuse est requise, le programmeur doit assurer l'intégrité des opérations via l'utilisation de routines de synchronisations). Ainsi, durant la phase itérative de l'approche collective, la taille des tampons temporaires est identique que ce soit en écriture ou en lecture (bien que celle-ci soit paramétrable, elle est de 4Mo par défaut).

Asynchronisme et fonction *split* : *ROMIO* fournit deux types de primitive d'accès asynchrone : pour les opérations au niveau d'un processus (`MPI_FILE_IXXX`) elles sont relativement similaires aux procédures usuelles d'UNIX. Dans le cadre de traitement collectif, le concept d'opération *split* est exploité. L'opération collective de type `MPI_FILE_XXX_ALL` se scinde alors en deux appels différés aux méthodes `MPI_FILE_XXX_BEGIN` et `MPI_FILE_XXX_END`. Les spécifications *MPI* impose comme unique contrainte l'achèvement de l'opération collective à la fin de la fonction `MPI_FILE_XXX_END`. Plusieurs implémentations sont alors possibles : l'opération collective peut être réalisée entièrement durant le premier appel ou bien uniquement lors de l'appel final. Une autre politique consiste à exécuter l'accès entre les deux routines dans un *thread* en arrière plan. Comme, nous l'avons cité dans le paragraphe précédent, [DT99] a montré l'importance de la répartition des étapes lors des opérations collectives : l'exécution totale de l'opération en arrière plan s'avère souvent pire que la réalisation au sein d'un processus principale. Il conclue en montrant qu'il est intéressant de réaliser seulement l'opération concrète d'entrée sortie dans un *thread* distinct. Toutefois, pour des raisons de fiabilité, *ROMIO* s'appuie seulement sur les routines asynchrones des systèmes de fichiers sous-jacent. Ainsi, lorsque ces dernières ne sont pas fournies, *ROMIO* fait appel aux routines synchrones[33]. De plus, le standard *MPI I/O* mentionne l'interdiction de faire appel à deux primitives de type *BEGIN* sur un même fichier avant que l'appel à la fonction *END* ait été réalisée. L'intégrité des données pouvant être mises en cause.

Consistance des données : La consistance au sein du modèle *MPI I/O* est définie comme «relâchée» ; une écriture réalisée par un processus est visible par tout autre processus lorsque ces derniers ont tous fait appel à une opération de synchronisation[34]. Un mode dit «*atomique*» permet d'offrir une consistance plus stricte au sein d'un communicateur ; si celui-ci est activé et que les processus appartiennent au même communicateur, toute modification des données par l'un sera immédiatement vu par les autres. Par défaut si le système de fichiers sous-jacent assure une cohérence forte alors *MPI I/O* assurera une consistance identique. Cependant, si *MPI I/O* complète l'architecture avec divers niveaux de caches (présent au dessus de l'*ADIO*), il sera nécessaire de maintenir la cohérence. Comme le spécifie le standard *MPI I/O* , l'atomicité au sein de *ROMIO* est donc «relâchée». Ainsi lorsque plusieurs processus réalisent des accès concurrents, le résultat final est indéfini. L'utilisation du mode *atomique* implique la mise en œuvre de «verrous» (*locks*) et donc de points de synchronisation implicites. Les systèmes de fichiers modernes permettent de choisir généralement entre «ces deux modes» : le mode «non atomique» étant habituellement moins coûteux et plus performant. Une définition judicieuse des communicateurs *MPI* évitera par conséquent l'activation du mode *atomique*.

Primitives de contrôle : L'interface *MPI I/O* est complété par des primitives permettant de faire passer des informations au couche basse et ainsi exploiter de manière optimale l'architecture logicielle et matérielle. *ROMIO* supporte quelques primitives prédéfinis permettant de spécifier différentes valeurs pour l'*ADIO* (nombre d'unités de *stripping*, taille des tampons, ...). De même, le programmeur a la possibilité de récupérer ces informations via l'utilisation de méthodes de type `MPI_File_Get_Info()` : le paramètre de la taille des *strippes* est par exemple relativement important afin de définir au mieux la taille des tampons intermédiaire exploités dans le cadre du *Data sieving* ou de l'approche *Two phases*. Cependant, rappelons une nouvelle fois que l'exploitation de ce genre de primitives peut altérer considérablement la portabilité du code.

[33] L'exploitation de tâches d'arrière plan nécessitant une implémentation *thread safe* du standard *MPI*.

[34] Si le processus A écrit et que B souhaite récupérer ces informations ; A et B doivent se synchroniser (MPI_File_Sync).

ROMIO* et *PVFS

Parmi le grand nombre de systèmes de fichiers parallèles suggéré à la communauté scientifique, le projet *PVFS* (cf. 2.2.4) a été relativement bien accepté. Plusieurs travaux ont été par conséquent mis en œuvre afin de proposer une couche *MPI I/O* au dessus de ce système, citehakan :mpiiopvfs,ilroy :mpiiopvfs,ching :noncontigousio,ching :noncontigousiob. C'est ces différentes améliorations que nous allons traitées tout au long de ce paragraphe...

Chapitre 4

Conlusion

Comme le montre, [TBDG02], l'utilisation «irréfléchi» de l'interface *MPI I/O* peut s'avérer néfaste

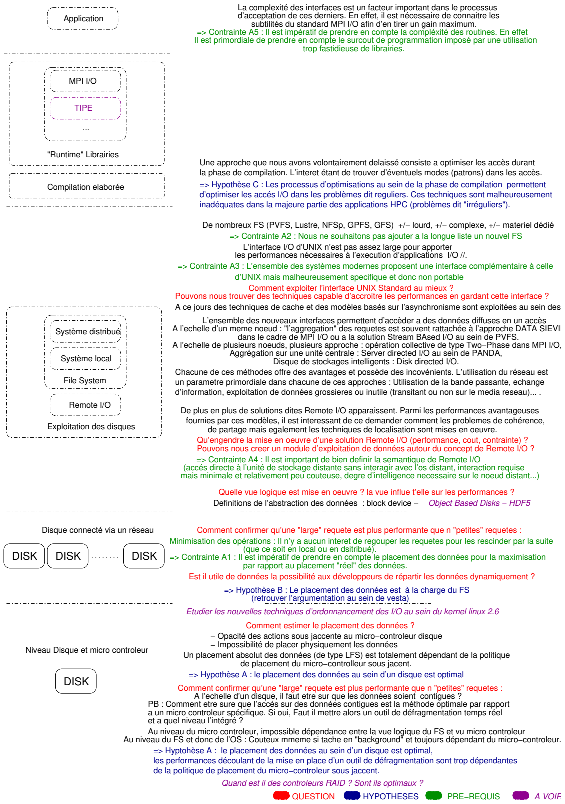

FIG. 4.1 – *Pile I/O*

Bibliographie

[ADN+95] Thomas E. Anderson, Michael D. Dahlin, Jeanna M. Neefe, David A. Patterson, Drew S. Roselli, and Randolph Y. Wang. Serverless network file systems. *Computer Science Division, University of California at Berkeley, CA 94720*, 1995.

[amd67] *Validity of the single-processor approach to achieving large scale computing capabilities*, 1967.

[AUB+96] Anurag Acharya, Mustafa Uysal, Robert Bennett, Assaf Mendelson, Michael Beynon, Jeffrey K. Hollingsworth, Joel Saltz, and Alan Sussman. Tuning the performance of I/O intensive parallel applications. In *Proceedings of the Fourth Workshop on Input/Output in Parallel and Distributed Systems*, pages 15–27, Philadelphia, May 1996. ACM Press.

[BBS+94] Robert Bennett, Kelvin Bryant, Alan Sussman, Raja Das, and Joel Saltz. Jovian : A framework for optimizing parallel I/O. In *Proceedings of the Scalable Parallel Libraries Conference*, pages 10–20, Mississippi State, MS, October 1994. IEEE Computer Society Press.

[BCR93] Rajesh Bordawekar, Alok Choudhary, and Juan Miguel Del Rosario. An experimental performance evaluation of Touchstone Delta Concurrent File System. In *Proceedings of the 7th ACM International Conference on Supercomputing*, pages 367–376. ACM Press, 1993.

[BL94] Ralph M. Butler and Ewing L. Lusk. Monitors, messages, and clusters : the p4 parallel programming system, April 1994.

[Bra99] Peter J. Braam. File systems for clusters from a protocol perspective, 1999.

[CACR95] Phyllis E. Crandall, Ruth A. Aydt, Andrew A. Chien, and Daniel A. Reed. Input/output characteristics of scalable parallel applications. In *Proceedings of Supercomputing '95*, San Diego, CA, December 1995. IEEE Computer Society Press.

[CBH+94] Alok Choudhary, Rajesh Bordawekar, Michael Harry, Rakesh Krishnaiyer, Ravi Ponnusamy, Tarvinder Singh, and Rajeev Thakur. PASSION : parallel and scalable software for input-output. Technical Report SCCS-636, ECE Dept., NPAC and CASE Center, Syracuse University, September 1994.

[CF01] Peter F. Corbett and Dror G. Feitelson. The Vesta parallel file system. In Hai Jin, Toni Cortes, and Rajkumar Buyya, editors, *High Performance Mass Storage and Parallel I/O : Technologies and Applications*, chapter 20, pages 285–308. IEEE Computer Society Press and Wiley, New York, NY, 2001.

[CFF+01] Peter Corbett, Dror Feitelson, Sam Fineberg, Yarsun Hsu, Bill Nitzberg, Jean-Pierre Prost, Marc Snir, Bernard Traversat, and Parkson Wong. Overview of the MPI-IO parallel I/O interface. In Hai Jin, Toni Cortes, and Rajkumar Buyya, editors, *High Performance Mass Storage and Parallel I/O : Technologies and Applications*, chapter 32, pages 477–487. IEEE Computer Society Press and Wiley, New York, NY, 2001.

[CFKL96] Pei Cao, Edward W. Felten, Anna R. Karlin, and Kai Li. Implementation and performance of integrated application-controlled file caching, prefetching, and disk scheduling. *ACM Transactions on Computer Systems*, 14(4) :311–343, 1996.

[CFP+95] Peter F. Corbett, Dror G. Feitelson, Jean-Pierre Prost, George S. Almasi, Sandra Johnson Baylor, Anthony S. Bolmarcich, Yarsun Hsu, Julian Satran, Marc Snir, Robert Colao, Brian Herr, Joseph Kavaky, Thomas R. Morgan, and Anthony Zlotek. Parallel file systems for the IBM SP computers. *IBM Systems Journal*, 34(2) :222–248, January 1995.

[CK98] Matthew P. Carter and David Kotz. An implementation of the Vesta parallel file system API on the Galley parallel file system. Technical Report PCS-TR98-329, Dept. of Computer Science, Dartmouth College, April 1998.

[CLG+94] Peter M. Chen, Edward K. Lee, Garth A. Gibson, Randy H. Katz, and David A. Patterson. RAID : high-performance, reliable secondary storage. *ACM Computing Surveys*, 26(2) :145–185, June 1994.

[CLRT00] Philip H. Carns, Walter B. Ligon III, Robert B. Ross, and Rajeev Thakur. PVFS : A parallel file system for linux clusters. In *Proceedings of the 4th Annual Linux Showcase and Conference*, pages 317–327, Atlanta, GA, October 2000. USENIX Association.

[CWS+97] Yong Cho, Marianne Winslett, Mahesh Subramaniam, Ying Chen, Szu wen Kuo, and Kent E. Seamons. Exploiting local data in parallel array I/O on a practical network of workstations. In *Proceedings of the Fifth Workshop on Input/Output in Parallel and Distributed Systems*, pages 1–13, San Jose, CA, November 1997. ACM Press.

[dBC93] Juan Miguel del Rosario, Rajesh Bordawekar, and Alok Choudhary. Improved parallel I/O via a two-phase run-time access strategy. In *Proceedings of the IPPS '93 Workshop on Input/Output in Parallel Computer Systems*, pages 56–70, Newport Beach, CA, 1993. Also published in Computer Architecture News 21(5), December 1993, pages 31–38.

[DT99] Phillip Dickens and Rajeev Thakur. Improving collective I/O performance using threads. In *Proceedings of the Joint International Parallel Processing Symposium and IEEE Symposium on Parallel and Distributed Processing*, pages 38–45, April 1999.

[DT01] Phillip M. Dickens and Rajeev Thakur. Evaluation of collective I/O implementations on parallel architectures. *Journal of Parallel and Distributed Computing*, 61(8) :1052–1076, August 2001.

[FK90] I. Foster and C. Kesselman. The grid : Blueprint for a new computing infrastructure. *eds. Morgan Kaufmann*, 1990.

[FWNK96] Samuel A. Fineberg, Parkson Wong, Bill Nitzberg, and Chris Kuszmaul. PMPIO— a portable implementation of MPI-IO. In *Proceedings of the*

Sixth Symposium on the Frontiers of Massively Parallel Computation, pages 188–195. IEEE Computer Society Press, October 1996.

[GAD+94] A. Geist, A.Beguelin, J. Dongarra, W. Jiang, R. Manchek, and V. S. Sunderam. *PVM : Parallel Virtual Machine : A Users' Guide and Tutorial for Networked Parallel Computing*. MIT Press, Cambridge, MA, USA, 1994.

[GH01] Emin Gabrielyan and Roger D. Hersch. Sfio a striped file i/o library for mpi. In *Eighteenth IEEE Symposium on Mass Storage Systems in cooperation with the Ninth NASA Goddard Conference on Mass Storage Systems and Technologies*, April 2001.

[GLDS96] William Gropp, Ewing Lusk, Nathan Doss, and Anthony Skjellum. High-performance, portable implementation of the MPI Message Passing Interface Standard. *Parallel Computing*, 22(6) :789–828, 1996.

[Gro01] Cilk-Cilk Is Supercomputing Technologies Group. *Cilk 5.3 – Reference Manual*, November 2001.

[GS93] William D. Gropp and Barry Smith. Chameleon parallel programming tools users manual, March 1993.

[HER+95] Jay Huber, Christopher L. Elford, Daniel A. Reed, Andrew A. Chien, and David S. Blumenthal. PPFS : A high performance portable parallel file system. In *Proceedings of the 9th ACM International Conference on Supercomputing*, pages 385–394, Barcelona, July 1995. ACM Press.

[HGFW02] Harry Hulen, Otis Graf, Keith Fitzgerald, and Richard W. Watson. Storage area networks and the high performance storage system, April 2002.

[HJMY00] Richard Hedges, Terry Jones, John May, and R. Kim Yates. Performance of an MPI-IO implementation using third-party transfer. In *IEEE Symposium on Mass Storage Systems*, pages 75–88, 2000.

[HO93] John H. Hartman and John K. Ousterhout. Zebra striped network file system. *Computer Science Division, Electrical Engineering and Computer Sciences, University of California at Berkeley, CA 94720*, 1993.

[HP96] J. L. Hennessy and D. A. Patterson. Computer architecture : A quantitative approach, 1996.

[IRU01] Jonathan Ilroy, Cyrille Randriamaro, and Gil Utard. Improving MPI-I/O performance on pvfs. August 2001.

[JMM+96] Terry Jones, Richard Mark, Jeanne Martin, John May, Elsie Pierce, and Linda Stanberry. An MPI-IO interface to HPSS. In *Proceedings of the Fifth NASA Goddard Conference on Mass Storage Systems*, pages I :37–50, September 1996.

[Kot94] David Kotz. Disk-directed I/O for MIMD multiprocessors. In *Proceedings of the 1994 Symposium on Operating Systems Design and Implementation*, pages 61–74. USENIX Association, November 1994. Updated as Dartmouth TR PCS-TR94-226 on November 8, 1994.

[Kot01] David Kotz. Disk-directed I/O for MIMD multiprocessors. In Hai Jin, Toni Cortes, and Rajkumar Buyya, editors, *High Performance Mass Storage and Parallel I/O : Technologies and Applications*, chapter 35, pages 513–535. IEEE Computer Society Press and John Wiley & Sons, 2001.

[KWC+97] S. Kuo, M. Winslett, Y. Chen, Y. Cho, M. Subramaniam, and K. Seamons. Application experience with parallel input/output : Panda and the H3expresso black hole simulation on the SP2. In *Proceedings of the Eighth SIAM Conference on Parallel Processing for Scientific Computing*, 1997.

[LD02] Pierre Lombard and Yves Denneulin. nfsp : A distributed nfs server for cluster of workstations. April 2002.

[Leb02] Adrien Lebre. Composition de service de données et de méta-données dans un système de fichiers distribué. Master's thesis, Joseph Fourrier University at Grenoble (France), June 2002.

[LR96] W. B. Ligon and R. B. Ross. Implementation and performance of a parallel file system for high performance distributed applications. In *Proceedings of the Fifth IEEE International Symposium on High Performance Distributed Computing*, pages 471–480. IEEE Computer Society Press, August 1996.

[MCFX97] Sachin More, Alok Choudhary, Ian Foster, and Ming Q. Xu. MTIO : a multi-threaded parallel I/O system. In *Proceedings of the Eleventh International Parallel Processing Symposium*, pages 368–373, April 1997.

[MJoMW03] Xiasong Ma, Xiangmin Jiao, and Michael Campbell oand Marianne Winslett. Flexible and efficient parallel I/O for large-scale multicomponent simulations. In *Proceedings of the Fourth Workshop on Parallel and Distributed Scientific and Engineering Computing with Applications*. IEEE Computer Society Press, April 2003.

[MS94] Steven A. Moyer and V. S. Sunderam. PIOUS : a scalable parallel I/O system for distributed computing environments. In *Proceedings of the Scalable High-Performance Computing Conference*, pages 71–78, 1994.

[MS95] S. Moyer and V. S. Sunderam. Parallel I/O as a parallel application. *International Journal of Supercomputer Applications*, 9(2) :95–107, Summer 1995.

[MWLY02] Xiaosong Ma, Marianne Winslett, Jonghyun Lee, and Shengke Yu. Faster collective output through active buffering. In *Proceedings of the International Parallel and Distributed Processing Symposium*, 2002.

[MWLY03] Xiasong Ma, Marianne Winslett, Jonghyun Lee, and Shengke Yu. Improving MPI IO output performance with active buffering plus threads. In *Proceedings of the International Parallel and Distributed Processing Symposium*. IEEE Computer Society Press, April 2003.

[NK97] Nils Nieuwejaar and David Kotz. The Galley parallel file system. *Parallel Computing*, 23(4) :447–476, June 1997.

[NKP+96] Nils Nieuwejaar, David Kotz, Apratim Purakayastha, Carla Schlatter Ellis, and Michael Best. File-access characteristics of parallel scientific workloads. *IEEE Transactions on Parallel and Distributed Systems*, 7(10) :1075–1089, October 1996.

[NL97] Bill Nitzberg and Virginia Lo. Collective buffering : Improving parallel I/O performance. In *Proceedings of the Sixth IEEE International Symposium on High Performance Distributed Computing*, pages 148–157, Portland, OR, August 1997. IEEE Computer Society Press.

[PBB+00] Kenneth W. Preslan, Andrew Barry, Jonathan Brassow, Russel Catalan, Adam Manthei, Erling Nygaard, Set Van Oort, David Teigland, Mike Tilstra, and Matthew T. O'Keefe. Implementing journaling in a linux shared disk file system. *in the 8th NASA Goddard Conference on Mass Storage Systems and Technologies in cooperation with the 7th IEEE Symposium on Mass Storage Systems*, 2000.

[PJS+94] B. Pawlowski, C. Juszczak, P. Staubach, C. Smith, D. Lebel, and D. Hitz. NFS version 3, design and implementation. In *Proceedings of the USENIX Summer 1994 Conference*, pages 65–79, Juin 1994.

[PTH+00] Jean-Pierre Prost, Richard Treumann, Richard Hedges, Alice Koniges, and Alison White. Towards a high-performance and robust implementation of mpi-io on top of gpfs. pages 1253 – 1262, September 2000.

[PTH+01] Jean-Pierre Prost, Richard Treumann, Richard Hedges, Bin Jia, and Alice Koniges. Mpi-io/gpfs, an optimized implementation of mpi-io on top of gpfs. In *Proceedings of the 2001 ACM/IEEE conference on Supercomputing (CDROM)*, pages 17–17. ACM Press, 2001.

[Raj02] Kumaran Rajaram. Principal design criteria influencing the performance of a portable, high performance parallel I/O implementation. Master's thesis, Department of Computer Science, Mississippi State University, May 2002.

[Sch02] Roger L.Haskin Frank B. Schmuck. Gpfs : A shared-disk file system for large computing clusters. *In Proceedings of the 5th Conference on File and Storage Technologies*, January 2002.

[Sch03] Phil Schwan. Lustre : Building a file system for 1,000-node clusters. In *Proceedings of the Linux Symposium, Ottawa*, July 2003.

[SCJ+95] K. E. Seamons, Y. Chen, P. Jones, J. Jozwiak, and M. Winslett. Server-directed collective I/O in Panda. In *Proceedings of Supercomputing '95*, San Diego, CA, December 1995. IEEE Computer Society Press.

[SL03] Jeffrey M. Squyres and Andrew Lumsdaine. A component architecture for lam/mpi. In *Proceedings, Euro PVM/MPI*, October 2003.

[SS00] K. Stockinger and E. Schikuta. Vimpios : A truly portable mpi-io implementation, 2000.

[SSD+94] Anthony Skjellum, Steven G. Smith, Nathan E. Doss, Alvin P. Leung, and Manfred Morari. The design and evolution of Zipcode. *Parallel Computing*, 20(4) :565–596, 1994.

[TBDG02] Juan Touriño, Jacobo Barro, Ramón Doallo, and Víctor M. Gulías. Performance analysis of mpi-i/o primitives on a pc cluster. In *Proceedings of the 2002 ACM symposium on applied computing*, pages 907 – 912, 2002.

[TGL96] Rajeev Thakur, William Gropp, and Ewing Lusk. An abstract-device interface for implementing portable parallel-I/O interfaces. In *Proceedings of the Sixth Symposium on the Frontiers of Massively Parallel Computation*, pages 180–187, October 1996.

[TGL98] Rajeev Thakur, William Gropp, and Ewing Lusk. A case for using MPI's derived datatypes to improve I/O performance. In *Proceedings of SC98 : High Performance Networking and Computing*. ACM Press, November 1998.

[TGL99a] Rajeev Thakur, William Gropp, and Ewing Lusk. Data sieving and collective I/O in ROMIO. In *Proceedings of the Seventh Symposium on the Frontiers of Massively Parallel Computation*, pages 182–189. IEEE Computer Society Press, February 1999.

[TGL99b] Rajeev Thakur, William Gropp, and Ewing Lusk. On implementing MPI-IO portably and with high performance. In *Proceedings of the Sixth Workshop on Input/Output in Parallel and Distributed Systems*, pages 23–32, May 1999.

[TGL02] Rajeev Thakur, William Gropp, and Ewing Lusk. Optimizing noncontiguous accesses in MPI-IO. *Parallel Computing*, 28(1) :83–105, January 2002.

[Tho96] Joel T. Thomas. The Panda array I/O library on the Galley parallel file system. Technical Report PCS-TR96-288, Dept. of Computer Science, Dartmouth College, June 1996. Senior Honors Thesis.

[VKS02] Murali Vilayannur, Mahmut Kandemir, and Anand Sivasubramaniam. Kernel-level caching for optimizing i/o by exploiting inter-application data sharing. September 2002.

[VSK+03] Murali Vilayannur, Anand Sivasubramaniam, Mahmut Kandemir, Rajeev Thakur, and Robert Ross. Discretionary caching for i/o on clusters. May 2003.

[WC95] Richard W. Watson and Robert A. Coyne. The parallel I/O architecture of the high-performance storage system (HPSS). In *Proceedings of the Fourteenth IEEE Symposium on Mass Storage Systems*, pages 27–44. IEEE Computer Society Press, September 1995.

[whi02] Lustre : A scalable, high performance file system, November 2002.