Laboratoire ID

Service de régulation et d'ordonnancement de requêtes d'Entrées/Sorties au sein des architectures parallèles

Thanh-Trung Van

sous la direction de
Adrien Lebre et Yves Denneulin

Membres du jury :
Joëlle Coutaz
Yves Denneulin
Jérôme Euzenat
Cyril Labbé

Juin 2005

Service de régulation et d'ordonnancement de requêtes d'Entrées/Sorties au sein des architectures parallèles

Thanh-Trung Van

sous la direction de
Adrien Lebre et Yves Denneulin

{Thanh-Trung.Van, Adrien.Lebre, Yves.Denneulin}@imag.fr
Laboratoire ID-IMAG
Montbonnot St Martin, Isère

Résumé

Composante clé d'un système informatique, la gestion de données suscite depuis fort longtemps l'attention de la communauté scientifique. La mise en œuvre de solutions logicielles capables de palier à la disparité de performance entre les unités de calculs (puissance des *CPUs*) et les sous-systèmes de stockage (temps d'accès) a toujours été primordiale. Par ailleurs, un grand nombre d'applications scientifiques (biologie, climatologie, physique des particules …) profitent de la forte présence des nouvelles architectures parallèles pour exécuter des problèmes complexes manipulant des quantités de données de plus en plus conséquentes. Pour tenter d'améliorer les temps d'accès et de traitement de ces informations, plusieurs solutions ont été proposées (systèmes de fichiers distribués, parallèles, librairies d'entrées/sorties, ordonnancement basé sur le mouvement des données …). Ces systèmes sont souvent complexes à utiliser et reposent sur des architectures matérielles dédiées qui peuvent s'avérer rapidement onéreuses (technologie *RAID*, fibre optique …).

Après un bref aperçu des solutions disponibles, nous présentons dans ce rapport le travail effectué autour d'une approche développée au laboratoire ID-IMAG depuis un peu plus d'un an. Fondée sur les routines du standard POSIX (`open/read/write/close`), aIOLi est une librairie d'Entrées/Sorties pour grappe. La première version de cette solution est utilisée afin de traiter de manière efficace les accès aux données au sein d'un même noeud. Le prototype émanant des travaux réalisés lors de mon master a pour but de réguler et d'ordonnancer les accès en provenance de plusieurs noeuds d'une grappe. Les résultats sont intéressants et s'avèrent prometteurs afin d'étendre le modèle à l'échelle des grilles. Nous verrons également les nombreux points en suspend et les perspectives qui en découlent.

Mots clés : grappes, système de fichiers, NFS, *Parallel I/O*, *High Performance Computing*, *MPI I/O*.

Table des matières

Remerciements **5**

Introduction **7**

I Etat de l'art **9**

1 Notions fondamentales **11**
1.1 Caractérisation des E/S dans les applications *HPC* 11
1.1.1 Généralités 11
1.1.2 Disparité entre données en mémoire et fichiers disques 12
1.1.3 Classification des applications 12
1.1.4 Type d'accès 13
1.2 Méthodes d'agrégation 13
1.2.1 "*List I/O*" 14
1.2.2 "*Data Sieving*" 14
1.2.3 "*Collective I/O*" 15
1.3 Ordonnancement dans les E/S 16
1.3.1 Notions élémentaires 16
1.3.2 Ordonnancement réactif 17
1.3.3 Ordonnancement anticipé 18

2 Systèmes existants **21**
2.1 Systèmes de fichiers distribués/parallèles 21
2.1.1 Généralités 21
2.1.2 NFS Version 4 22
2.1.3 Clusterfile 24
2.2 Librairies d'E/S parallèles 25
2.2.1 Panda 25
2.2.2 ROMIO 27
2.3 Gestion et de transfert à l'échelle des grilles 27
2.3.1 GridFTP : Protocole de transfert de données des grilles 28

II aIOLi **31**

3 aIOLi : gestion des E/S parallèles **33**
3.1 Présentation du système aIOLi 33
3.2 Coordination entre plusieurs SMPs 34
3.2.1 Architectures logicielles 35
3.3 Synchronisation des requêtes 36
3.3.1 Exclusion mutuelle distribuée 37

3.3.2 Les diverses solutions . . . 37
3.3.3 Solution retenue . . . 38
3.4 Ordonnancement de requêtes . . . 40
3.4.1 Choix du critère dans aIOLi . . . 40
3.4.2 Algorithmes d'ordonnancement proposés . . . 41
3.4.3 Contrainte de sous-requêtes . . . 43

4 Implantation et Evaluation **47**
4.1 Implantation du prototype . . . 47
4.2 Expérimentations . . . 47
4.3 Évaluation dans le cadre mono-applicatif . . . 48
4.3.1 Détection des schémas d'accès parallèles . . . 48
4.3.2 Phénomène de décalage . . . 49
4.3.3 Problème de prédiction . . . 49
4.3.4 Résultats . . . 50
4.4 Évaluation dans le cadre multi-applicatif . . . 51

5 Bilan **53**

6 Conclusion **55**

Table des figures

1.1 Accès *simple-strided* . . . 13
1.2 Exemple d'écriture utilisant l'approche"*List I/O*" . . . 14
1.3 Ordonnancement pour réduire le temps de recherche sur disque - SPTF. . . . 18
1.4 SPTF devient de type FCFS. . . . 19
1.5 Ordonnanceur de partage proportionnel . . . 19

2.1 NFS . . . 22
2.2 Architecture d'un système pNFS . . . 23
2.3 Distribution des données dans Clusterfile . . . 24
2.4 Architecture de Panda . . . 25
2.5 Architecture de ROMIO . . . 27
2.6 Niveaux d'optimisations dans ROMIO . . . 28

3.1 Un master par serveur d'E/S . . . 35
3.2 Un master par sous-groupe . . . 36
3.3 Les accès vers un système de stockage distant . . . 36
3.4 Délai de synchronisation . . . 37
3.5 Algorithme simple pour le problème d'exclusion mutuelle . . . 38
3.6 Synchronisation des accès . . . 39
3.7 Algorithme WSJF . . . 42
3.8 Une variante de MLF . . . 44
3.9 Contrainte de sous requêtes . . . 45

4.1 Architecture du système aIOLi . . . 48
4.2 Requêtes décalées . . . 49
4.3 Décomposition d'un fichier de 2Go incluant 8 instances MPI . . . 50
4.4 Décomposition de 2 fichiers de 2Go incluant 4*2 instances MPI . . . 51
4.5 Décomposition de 2 fichiers de 2Go incluant 4*2 instances MPI . . . 52

Remerciements

Je tiens à remercier tout d’abord Monsieur Adrien Lebre, pour m’avoir encadré et pour m’avoir aidé chaleureusement tout au long de mon séjour au laboratoire ID.

Je tiens à remercier Monsieur Yves Denneulin, mon responsable de stage, et Madame Brigitte Plateau, directeur du laboratoire ID, pour m’avoir accueilli au sein de l’équipe MESCAL du laboratoire.

Je remercie tout le personnel du laboratoire ID (permanents, thésards et stagiaires) pour l’ambiance amicale et les conseils précieux qu’ils m’ont donnés.

Enfin, je tiens à remercier les rapporteurs et les membres du jury qui me font l’honneur de juger ce travail.

Introduction

Composante clé d'un système informatique, la gestion de données suscite depuis fort longtemps l'attention de la communauté scientifique. Que ce soit les systèmes de gestion de base de données, de partage de fichiers au sein d'un LAN ou encore à l'échelle d'Internet (applications "pair à pair"), ces systèmes ne cessent de s'adapter dans le but de proposer des solutions permettant d'exploiter le maximum des capacités des plate-formes sur lesquels ils sont déployés (puissance d'analyse, ressources importantes de stockage, débits ...) et ce afin de pallier à l'écart considérable de performance entre la puissance de calcul et les temps d'accès au données.
Cette disparité, déjà signalée en 1967, [1], et confirmer plus récemment [2], a été fortement accrue par l'avènement des architectures parallèles modernes visant à exploiter de manière plus ou moins uniforme plusieurs ordinateurs inter-connectés par un réseau efficace.
Ces nouveaux "super-calculateurs", communément appelé grappe, fournissent des puissances de calculs permettant à de nombreuses applications scientifiques (météorologie, océanographie, génomique, physique du nucléaire ...) d'observer (et/ou de simuler) de nouveaux problèmes et de repousser ainsi d'anciennes limites. Toutefois, la quantité considérable de données exploitée par ces nouveaux programmes requiert de la part de ces architectures, un système de gestion de données efficace et fiable. Le projet européen DataGrid, par exemple, traite plusieurs Peta octets par an (environ 10^{15} octets) lors des expérimentations de physique nucléaire. Il est donc primordiale de proposer des systèmes de stockages capable de bénéficier des avantages précédemment indiqués tout en prenant en compte des contraintes (tolérance aux pannes, passage à l'échelle, ...) liés à ce type d'architecture.

Les systèmes employés pour le partage de fichiers au sein d'un réseau local comme NFS ne permettent pas de répondre aux besoins des nouvelles applications. En effet, l'implication croissante des entrées/sorties[1] a imposé des nouvelles contraintes ainsi que des nouveaux modèles de programmation (accès concurrent à grains fins, multiplicité des accès non contigus ...). De nombreux travaux ont été et sont actuellement menés afin de proposer des solutions efficaces. Parmi ces travaux, deux axes majeurs se distinguent : le premier consiste à proposer un système de stockage complet tentant de prendre en compte aussi bien les aspects matériels que logiciels. Ces nouveaux systèmes de stockage, usuellement nommés "systèmes de fichiers parallèles" reposent sur plusieurs entités permettant une répartition de la charge. La seconde approche se concentre uniquement au niveau des routines d'accès en enrichissant les interfaces d'accès standard (`open/read/write/close`) afin de proposer une gestion plus fine des unités de stockages sous-jacentes.

Par ailleurs, l'extension des grappes aux grilles [3] (interconnexion de plusieurs architectures de calculs par un réseau dédié) augmentent les phénomènes de type goulot d'étranglement spécifiques aux systèmes de stockage. En effet, une application distribuée sur une grille peut accéder conjointement à plusieurs serveurs de stockages distants. Par conséquent, un système de stockage à l'échelle des grilles doit tenir compte des multiples propriétés des réseaux, des configurations et des principales fonctionnalités de chaque sous système en plus d'un nouveau changement d'échelle en terme de concurrence des accès. Toutefois, il est presque certain que le développement des applications futures va s'appuyer sur ces grandes disponibilités de puissance de calcul et d'espace de stockage quasi-infini ; la mise en œuvre d'outils performants est donc impérative afin de proposer des supports fiables et puissants permettant de faire interagir calculs et échanges d'informations.

[1] Par la suite, nous employerons l'acronyme E/S pour Entrées/Sorties.

Méthodes d'accès locales, partages de ressources au sein d'un LAN, au sein d'une grappe ou encore échange d'informations sur le web via les nombreux systèmes pair à pair comme Napster, edonkey ou encore BitTorrent ; la problématique du partage des données a toujours suscité un engouement certain. La large diversité des travaux de recherches menées rend très complexe voir impossible la rédaction d'un rapport recouvrant l'ensemble des problèmes propres à cette large thématique. Par conséquent, l'ensemble des efforts a été dirigé vers l'étude des Entrées/Sorties au sein des architectures et des applications parallèles fortement distribuées ; améliorer les critères de performances étant le but recherché. Nous présentons au sein de ce rapport, le travail effectué autour d'une solution développé au laboratoire ID depuis un peu plus d'un an. La première version de cette librairie est utilisé afin de traiter de manière efficace les E/S au sein d'un même nœud SMP[2]. Le prototype émanant de notre travail a pour but de réguler et d'ordonnancer les accès provenant de plusieurs nœuds d'une grappe (le modèle pouvant être étendu aux grilles). L'objectif étant de donner la meilleure utilisation des disques tout en garantissant l'équité et l'interactivité entre les applications.

Durant la première partie de ce rapport, nous abordons des notions fondamentales liés à la gestion des E/S au sein des applications parallèles *HPC*[3], chapitre 1. Différentes solutions proches de nos travaux sont présentées à la suite, chapitre 2. Le premier chapitre de la seconde partie s'attarde sur la solution aIOLi et décrit les aspects théoriques liés à une régulation distribuée des E/S (travaux proposé dans le cadre du master), chapitre 3. Le chapitre 4 présente les diverses expérimentations menées sur la grappe idpot[4] et un bilan générale sur les travaux réalisés est établi au chapitre 5.

Enfin, nous énoncerons, durant la conclusion, les diverses perspectives entrouvertes par l'étude menée et l'intérêt de la mise en œuvre d'une tel approche à l'échelle des grilles.

[2] *Symmetric MultiProcessor*, machine composée de plusieurs processeurs.
[3] High Performance Computing.
[4] `http:\\frontal38.imag.fr`

Première partie

Etat de l’art

Chapitre 1

Notions fondamentales

Avant d'entrer plus amplement dans le sujet, nous allons définir d'un point de vue formel ce que nous entendons par :

Application parallèle : une application multi-tâches (*threads* ou processus *SPMD*[1]) qui travaille en concurrence afin de résoudre un problème.

Calcul haute performance (*HPC*) : Exploitation d'architectures informatiques puissantes (grappes, grilles, super-calculateurs) afin de traiter dans un temps "raisonnable" des applications parallèle. Parmi ces applications, plusieurs s'appuient sur une grande quantité de données.

E/S Parallèle : Accès au données (lecture et/ou écriture) par une application de manière parallèle. Cette notion de parallélisme peut être amplifier selon l'architecture matérielle employé pour le stockage de données (plusieurs disques, plusieurs serveurs ...).

1.1 Caractérisation des E/S dans les applications *HPC*

Dans cette section, nous nous attardons sur divers travaux qui ont eu pour but l'étude des comportements et des modes d'accès utilisés dans les applications scientifiques hautement parallèles. Cette caractérisation a été abordée dans plusieurs ouvrages [4, 5, 6] et a permis de définir de manière non exhaustive l'ensemble des besoins E/S.

1.1.1 Généralités

Dans les systèmes de fichiers Unix/Linux (ext2, ext3, UFS ...), un fichier est une séquence d'octets. Chaque octet d'un fichier peut être adressé par un numéro d'offset qui indique la position de cet octet dans le fichier. Quand une application ouvre un fichier (`open`), le système d'exploitation retourne un numéro, le descripteur de fichier, qui représente ce fichier. Un pointeur de fichier, associé à ce descripteur, permet de se positionner au sein du fichier. Tous les opérations suivantes sur ce fichier au sein d'une même application sont exécutées en utilisant ce descripteur de fichier. Chaque fois qu'une application lit ou écrit (`read/write`), le système de fichier repositionne le pointeur de ce fichier en accord avec l'opération effectuée. L'utilisateur peut déplacer son pointeur en utilisant la fonction `lseek` afin d'accéder à une partie particulière des données. Plusieurs applications peuvent accéder au même fichier sur une même période et à différents endroits. Dans ce cas, il est important de définir la sémantique utilisée pour assurer la cohérence [7]. Les systèmes de fichiers Unix implémentent une cohérence forte : toute modification qui survient sur un bloc[2] est vue instantanément sur l'ensemble des nœuds accédant à l'information au même moment. De plus, la cohérence est séquentielle : le résultat de deux écritures concurrentes est similaire à celui de deux écritures séquentielles et ne peut être un "mélange" des deux opérations. Pour certaines applications, cette cohérence forte est requise, il est donc important d'essayer de proposer un système qui ne la dégrade pas.

[1] *Single Program Multiple Data*, un tâche MPI par exemple.

[2] Le terme de bloc équivaut à la granularité du système de fichiers. C'est la plus petite portion remise lors d'une requête.

Lorsque le fichier est fermé par l'application (`close`), les informations concernant l'accès au fichier sont perdues et ne pourront être réutilisées lors d'une prochain accès.

1.1.2 Disparité entre données en mémoire et fichiers disques

Une des premières difficultés consiste à tenir compte de la différence entre la distribution des données en mémoire et le placement physique de ces mêmes données sur le disque, [8]. En effet, au niveau d'une application, l' utilisateur travaille sur "fichiers" sans tenir compte du placement réel de ces derniers sur la ou les différentes zones de stockages (opposition entre l'abstraction "fichier" et les blocs disques). Cette différence peut être décrite par les définitions suivantes :

Structure de données globale ou logique : C'est la distribution des données en mémoire entre les nœuds de calculs (au sein d'une application).

Structure de données physique : C'est la distribution d'un fichier entre différents nœuds de stockage (*file layout*). Par exemple, la plupart des systèmes de fichiers parallèles utilisent la technique *data striping* qui consiste à découper un fichier en plusieurs morceaux (*striping unit*) et à les répartir sur différentes disques. Les données pouvant être alors accédées de manière parallèle.

Un unique accès au sein d'une application peut engendrer un nombre plus ou moins conséquent de transfert. La correspondance entre la structure de données globale et la structure de données physique joue en rôle important pour les performances globale du système.

1.1.3 Classification des applications

Bien que l'usage des entrées / sorties au sein des applications *HPC* soit variée, trois principales classes de comportement ont été définies [9] :

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 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 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[3] 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.

Parmi les paramètres utilisés pour l'évaluation des routines d'accès, le facteur de réactivité[4] s'avère être un critère fondamentale dans les E/S parallèles. Dans un contexte parallèle, ce critère est dépendant de la manière dont les applications accèdent aux données. Ce point est abordé dans la partie suivante.

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

[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).

1.1.4 Type d'accès

Pour améliorer la performance des E/S, nous devons identifier les différents types d'accès utilisés par les applications *HPC*. La liste ci après les dresse brièvement :

Accès séquentiel : Deux requêtes sont séquentielles si la dernière débute à un offset supérieur à celui où de la première requête s'est finie.

Accès consécutif : Deux requêtes sont consécutives si la dernière débute exactement à l'offset où la première s'est finie.

Accès disjoint régulier (*simple-strided*) : Une série de requêtes d'E/S est qualifier d'accès dit "*simple-strided*" si chaque requête s'effectue sur une même quantité de données et que la distance entre deux requêtes successives est identique (le *stride* est le nombre d'éléments entre deux requêtes successives). Une série d'accès "*simple-strided*" est appelée un "*strided segment*". Exemple : supposons que nous avons une matrice de taille 3x3 ; chaque colonne de la matrice est attribué à un processus et la matrice est stocké sur le disque par ligne. Dans ce cas chaque processus lit un élément dans une ligne puis bascule sur une autre ligne pour lire l'élément suivant. Les accès provenant d'un même processus forment un *strided segment*.

Accès disjoint de niveau n (*nested-strided*) : le modèle d'accès "*nested-strided*" est similaire au modèle "*simple-strided*" ; l'unique différence et qu'il se compose d'une série de "*strided segments*" séparées par une même distance dans le fichier.

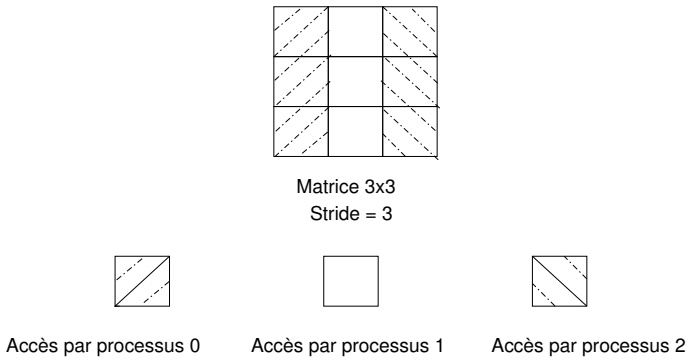


FIG. 1.1 – Accès *simple-strided*

Les accès recouvrants de type "*simple/nested-strided*"interviennent quand des données successives sont accédées par plusieurs processus de grappe. 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 alors qualifiés de recouvrant. L'emploi de technique de cache et de préchargement (*read ahead*) usuellement mise en œuvre dans le cadre de programme séquentielle ne s'apprêtent pas ou mal à ces nouveaux modes d'accès .

Dans les applications parallèles E/S intensive, les fichiers sont souvent grands et plusieurs nœuds accèdent de manière simultanée et non-contiguë aux données. Ce type de comportement est souvent fatal, entraînant une diminution brusque des performances d'E/S). Plusieurs techniques ont été suggérées afin de réduire cet impact. L'emploi de techniques comme l'agrégation basées sur les accès recouvrants par exemple, diminue nettement le temps de réactivité. Le ré-ordonnancement à la volée (*online*) est également une méthode qui s'applique dans certains cas. Ce sont ces différentes techniques que nous abordons dans la suite de ce chapitre.

1.2 Méthodes d'agrégation

Comme nous l'avons indiqué, une des raisons importante des limitations des systèmes d'E/S viens du fait que les applications envoient souvent plusieurs petites requêtes disjointes. Ce mode d'accès en-

gendre un premier surcoût lié au grand nombre de demandes circulant sur les divers canaux de transmission (bus/réseau de communications) mais de plus augmente considérablement le temps de traitement de ces dernières. En effet, il est important de rappeler que les disques utilisés pour le stockage des données traitent un large accès beaucoup plus vite qu'une requête de taille égale mais divisée en plusieurs petites demandes (un disque peut perdre jusqu'a 9ms pour se positionner et ne nécessite que 3ms pour transférer un bloc de 64Ko [10]). Pour résoudre ce premier problème, plusieurs méthodes dites "d'agrégation" ont été proposées. Le but de ces méthodes est d'agréger plusieurs petites requêtes non contiguës en de plus grandes réduisant ainsi les temps d'accès. Si cette agrégation concerne les requêtes provenant d'un même processus, une approche de type *"Data Sieving"* est préconisée. Si cette agrégation concerne les requêtes provenant de plusieurs processus, les méthodes collectives (*"Collective I/O"*) sont alors plus appropriées. Une dernière approche consiste à transmettre au sein d'une même requête système, un vecteur contenant plusieurs accès. C'est la première technique que nous allons présenter.

1.2.1 *"List I/O"*

L'approche "*List I/O*" [11], fournit des routines permettant d'indiquer au sein d'un unique appel plusieurs accès. Une liste de couple (offset, taille) décrit la distribution des données en mémoire et une liste similaire est utilisée pour effectuer la correspondance sur le disque. L'exemple 1.2 présente cette technique : "*mem_list_count*" est le nombre de régions non contiguës en mémoire, "*mem_offsets[]*" est le tableau des pointeurs vers le début de ces régions et "*mem_lengths[]*" un tableau contenant la longueur de chaque segment correspondant. De manière similaire, les paramètres "*file_list_count*", "*file_offsets[]*" et "*file_lengths[]*" décrivent la distribution physique.

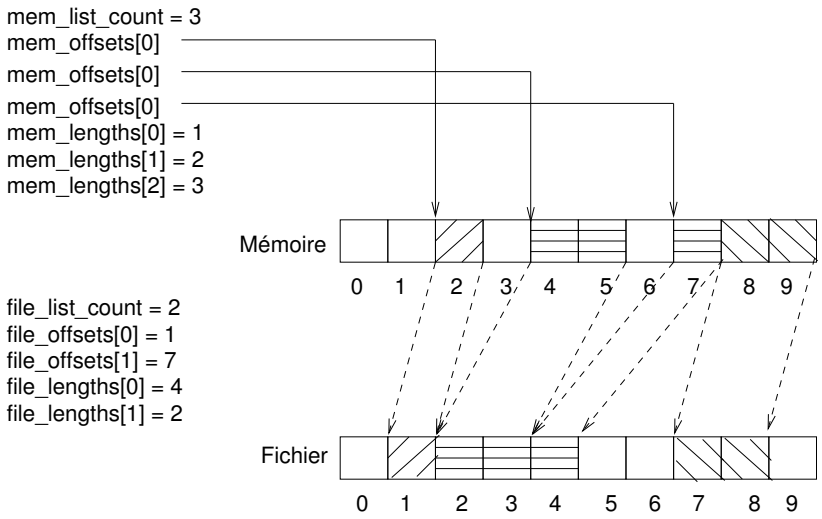


FIG. 1.2 – Exemple d'écriture utilisant l'approche"*List I/O*"
Un appel encapsule 4 requêtes.

Le principal inconvénient de cette approche est que le système de fichiers sous jacent doit proposer l'interface équivalent. Par exemple l'appel POSIX `readv()` se décompose en réalité en plusieurs sous requêtes `read()` transmises indépendamment au serveur.

1.2.2 *"Data Sieving"*

Le "*Data Sieving*" est une technique permettant d'agréger plusieurs petites requêtes non contiguës provenant d'un même processus. Chaque nouvelle requête obtenue recouvre à un accès de type *strided* ou *nested*. L'exemple ci-après illustre le principe de cette méthode : Supposons que nous avons 5 requêtes de lecture d'un même processus et que ces accès ne soient pas contigus : 5 commandes de lecture vont être

exécutées[5] ; dans la méthode "*Data Sieving*", la position de début et de fin de ces requêtes est calculée ; une unique opération de lecture est alors réalisée et la totalité des données est placée dans un tampon temporaire avant d'être redistribuer vers les diverses zones prévues par l'application. Une premier inconvénient de cette méthode est que la taille limitée des tampons temporaires en plus du transfert de données potentiellement inutiles (les données entre les "*strides*"). Par ailleurs, les écritures de type "*Data Sieving*" nécessitent des opérations de type "*read-modified-write*" afin de garder les données entre les "*strides*" cohérentes ; ce qui peut se révéler rapidement très coûteux.

1.2.3 *"Collective I/O"*

Les méthodes dites "*Collective I/O*" permettent d'agréger des requêtes émanant de plusieurs processus. Pour illustrer la nécessité de cette méthode, nous allons considérer l'exemple ci-après : supposons que nous avons une matrice de taille 4x4 et 4 processus, chaque processus veut accéder à une colonne de cette matrice. Si la matrice est stockée par ligne sur le disque, un processus doit exécuter 4 commandes de lecture, soit un total de 16 commandes de lecture différentes.

Si nous considérons maintenant le scénario suivant, chaque processus lit une ligne de cette matrice, puis ils redistribuent les données entre eux. La totalité de commandes de lecture est 4 et chacun des accès est plus conséquent et donc moins coûteux. Cet exemple montre la nécessité des méthodes de type "*Collective I/O*". Plusieurs approches ont été suggérées. Elles se distinguent par le "lieu physique" où l'opération collective est réalisée : si l'agrégation est exécuté entre les processus (sur les nœuds de calculs), la méthode la plus employé est l'approche "*Two-Phase I/O*" ; si l'agrégation est exécutée aux niveaux des disques, nous parlons de système "*Disk-Directed I/O*" enfin si l'approche est réalisé au sein d'un serveur, c'est la méthode "*server-directed I/O*". Nous présentons ces différents approches ci-après :

"Two-Phase I/O" Cette méthode [12], comme son nom l'indique, se compose de deux phases principales : après un consensus entre les processus participant, la première phase consiste à récupérer les données , la seconde concerne la redistribution de ces dernières entre chaque processus. Pour exécuter la première phase, chaque processus doit connaître les données nécessaire aux autres. L'avantage de cette méthode est qu'il permet de faire des accès larges et contigus et par conséquent de réduire fortement le temps de réactivité. Cependant, l'inconvénient de cette méthode est qu'elle requiert une fois encore des tampons supplémentaires requis lors de la phase de redistribution. De plus, dans cette méthode, les données sont transférées deux fois dans le réseau.

"Disk-Directed I/O" Dans la méthode "*disk-directed I/O*" [13], chaque processus envoie directement ses requêtes à tous les nœuds de stockages (serveur de fichiers). Les nœuds collectent ces requêtes et calculent les blocs de données correspondant, puis les ordonnent de manière séquentielle pour minimiser le temps de lecture/écriture en réduisant le mouvement de tête de lecture des disques. Les serveurs transfèrent les données aux tampons d'utilisateurs (ou à l'inverse dans le cas d'écriture).
La méthode "*Disk-Directed I/O*" a plusieurs avantages par rapport à l'approche "*Two-Phase I/O*" : les données sont transférées seulement une fois dans le réseau et ne requiert pas de tampons supplémentaires. Par ailleurs, son intervention à bas niveaux lui permet d'exploiter au mieux la structure de données physique (cf. 1.1.2) et de choisir un ordre séquentiel basé sur le placement réel des données. Cependant, cette méthode peut engendrer un grand nombre de petites messages pour transférer les données entre les serveurs et les tampons utilisateurs. Enfin, il est important de noter que cette méthode n'a pas encore été implémentée dans un système de fichiers ; ses performances ont été estimées seulement par des simulations.

"Server-Directed I/O" Dans cette méthode, les serveurs prennent en charge le contrôle des flux d'E/S afin de maximiser les accès séquentiels. A la différence de la méthode précédente et comme l'approche "*Two-Phase*", les techniques "*Server-Directed I/O*" se basent sur la structure logique des données pour ordonner et/ou agréger les requêtes. Un modèle similaire ordonnant les requêtes au sein d'un fichier a été implémenté au sein du projet PANDA, présenté dans la section 2.2.1.

[5] L'utilisation de la commande `readv` présnte sur les systèlmes POSIX engendre également 5 appels `read` sur les architectures Linux.

1.3 Ordonnancement dans les E/S

Les méthodes d'ordonnancement sont usuellement appliquées au sein des systèmes informatiques. L'avènement des architectures parallèles modernes comme les grappes a engendré une multitude de travaux ayant pour but l'optimisation de la gestion des ressources, principalement sur le positionnement des tâches (processus). Certaines des techniques proposées peuvent être appliquées dans le cadre de la gestion des E/S. Nous allons aborder ces aspects au sein de cette partie.

1.3.1 Notions élémentaires

Dans le vocabulaire lié à l'ordonnancement et par analogie, nous dirons qu'une **requête** correspond à un **travail** (ou une **tâche**) et que le **serveur de données** correspond à un **processeur**. Par ailleurs, Il est important de bien distinguer les problèmes dits **offlines** de ceux **onlines**. Dans un modèle dit *offline*, tous les travaux sont connus d'avance et le choix d'un ordonnancement efficace pour un critère donné est donc facilité. Au contraire, dans un modèle *online*, les travaux (requêtes) arrivent dans le temps et le système d'ordonnancement ne connaît qu'un travail quand il est arrivée. De plus, les problèmes *online* se divisent en deux sous parties : *clairvoyants* et *non-clairvoyants*. Un algorithme *clairvoyant* a connaissance du temps d'exécution d'un travail dès l'arrivée de celui-ci dans la queue. Au contraire, un algorithme *non-clairvoyant* ne connaît le temps d'exécution d'une tâche que quand celle-ci est finie. Par exemple, un serveur web qui fournit des documents statiques peut être modélisé par le modèle *online clairvoyant* puisque les tailles des documents sont connus d'avance. Par contre, un serveur web qui fournit des pages dynamiques (ASP, JSP ...) ne peut être modélisé que par un modèle *online non-clairvoyant*.

Critères d'évaluation

Plusieurs critères peuvent être utiliser pour évaluer un algorithme d'ordonnancement, les plus courants dans les problèmes *offline* quand les travaux arrivent en *batch* sont :

Le "makespan" : la "date" de finition de tous les travaux.

La somme (ou totalité) des temps de finition : la somme des temps de finition de tous les travaux, *total completion time*.

Pour les algorithmes *online* quand les travaux arrivent dans le temps, d'autres critères sont employées tel que :

La somme des temps de réponse le temps de réponse d'un travail est la différence entre son temps de finition et son temps d'arrivée. La totalité des temps de réponse est la somme de tous les temps de réponse des travaux, *total response time*[6].

La somme de "stretch" : le *stretch* d'un travail est définit de manière suivante :

$$stretch = \frac{temps\ de\ reponse}{temps\ de\ traitement} \tag{1.1}$$

Le *stretch* est une mesure pour exprimer le temps d'attente d'un travail (ou d'un utilisateur).

Algorithmes d'ordonnancement *online* usuels

Cette partie dresse un bilan très succinct des approches les plus connues pour les problèmes d'ordonnancement *online*, [14] :

SRPT : *Shortest Remaining Processing Time*, cet algorithme exécute toujours les travaux avec le temps restant le plus petit.

FIFO : *First In First Out*, cet algorithme est aussi appelé *First Come First Served*. Il exécute toujours le travail qui arrive le plus tôt.

SJF : *Shortest Job First*, cet algorithme exécute toujours le travail ayant la taille initiale la plus petite. C'est un algorithme efficace pour minimiser la totalité de temps d'exécution.

[6] Connu parfois sous le nom de *total flow time*.

Les algorithmes suivants sont plus spécifiques aux problèmes non-clairvoyants :

RR : *Round Robin*, cet algorithme distribue de manière équitable les ressources à tous les travaux. Chaque travail est exécuté pendant un quantum de temps avant qu'un autre travail dans la liste commence à exécuter. Les nouveaux travaux sont placés à la fin de la liste de travaux.

SETF : *Shortest Elapsed Time First*, cet algorithme exécute le travail qui a été exécuté le moins. Plusieurs recherches, [15], [16], ont montré que SETF est un bon algorithme pour minimiser la totalité de temps de réponse. Il existe plusieurs variantes de cet algorithme, [17], [18].

MLF : *Multilevel Feedback*, c'est une des variantes de l'algorithme SETF. Décrite dans [19], cette méthode est généralement celle employé dans les systèmes d'exploitation pour l'ordonnancement des processus. Il utilise un ensemble de queues, chaque queue a une propre priorité et un temps de traitement spécifié. Chaque fois qu'un nouveau travail est créé, il est inséré dans la queue qui a la priorité la plus élevée. Si le temps alloué lors de son premier passage n'est pas suffisant, il est alors inséré dans une queue de priorité moindre mais de temps de traitement plus élevé. Les queues sont traitées selon leurs ordres de priorité. Une queue ayant la propriété basse est traité seulement quand tous les queues qui ont les propriétés plus élevées sont vidés.

1.3.2 Ordonnancement réactif

L'ordonnancement réactif ("*reactive scheduling*") est une méthode visant à proposer un ordonnancement adéquat selon la charge et le comportement des systèmes de stockages. Habituellement, un système d'E/S propose un ordonnancement particulier pour les requêtes (par exemple *FIFO*) et ce quelque soit la charge du système. Cette stratégie "fixe" ou "figée' ne permet pas une gestion fine des E/S et de plus, se révèle être inapproprié dans un contexte d'E/S parallèles stressant énormément les systèmes de stockages. Décrit dans [20], la méthode d'ordonnancement réactif est basée sur une sélection dynamique de politiques d'ordonnancement. Elle utilise les informations sur l'état et charge du système pour déterminer un ordre d'exécution de requêtes.
Ce système se compose de trois parties principales suivantes :

Un ensemble d'algorithmes d'ordonnancement : Les algorithmes d'ordonnancement nous donnent un ordre d'exécution des requêtes du système. Un algorithme d'ordonnancement peut être basé sur l'ordre d'arrivée de requête (tel que FIFO [7]) ou basé sur le temps de délai entre les deux requêtes contiguës (tel que *Shortest Seek Time First*).

Un modèle du système : Le modèle du système est en charge de prévoir les performances du système (temps d'exécution de requêtes) en utilisant des paramètres tels que le débit de disque, le débit de réseau, la charge des travaux, la stratégie d'ordonnancement mise en œuvre ... Par exemple, le temps d'exécution d'une requête de lecture peut être modélisé par la formule suivante dans un premier temps :

$$T_{taskread} = T_{over} + E_{dist} E_{req} T_{ioread} \tag{1.2}$$

T_{over} : Le temps de service lié à la gestion de l'opération d'E/S (envoi/reception /analyse de la requête).
E_{dist} : L'efficacité d'accès basée sur la distribution de données de cette requête. Par exemple, si un accès concerne des parties non contiguës sur le disque, il est "pire" qu'un autre accès contenant des parties contiguës.
E_{req} : L'efficacité d'accès en tenant compte des caractéristiques de l'ensemble des requêtes en cours. Par exemple les accès non contigus vont être plus coûteux que les accès contigus.
T_{ioread} : Temps de lecture de données. Ce temps doit être calculé en considérant les influences du cache.
Le but de l'approche est de déterminer le meilleur ordonnancement, il est primordiale d'intégrer à la

[7] ou *First Come First Served*

précédente formule un ratio selon la stratégie mise en œuvre : E_{opt} représente cette influence. La formule devient alors :

$$T_{taskread} = T_{over} + E_{opt}E_{dist}E_{req}T_{ioread} \tag{1.3}$$

Les paramètres T_{over}, E_{opt}, E_{req}, E_{dist}, T_{ioread} sont estimés en tenant compte de l'état du système actuel.

Un mécanisme de sélection : Ce composant s'appuie sur le modèle du système afin de trouver un politique d'ordonnancement adéquat selon l'état actuel du système. La stratégie ainsi déterminée est utilisée pour sélectionner une requête à exécuter.

Cette méthode a été évalué au sein du système de fichiers PVFS 1 (*Parallel Virtual File System*) [21]. Elle a permis d'observer que selon l'état du système, la performance peut varier de 6% à 60% selon l'algorithme d'ordonnancement utilisé. Une sélection dynamique du meilleur algorithme s'avère donc très intéressante.

1.3.3 Ordonnancement anticipé

Les stratégies d'ordonnancement employées à bas niveau(*disk scheduler*) exploitent souvent des approches type de "*work-conserving*", c'est-à-dire qu'elles exécutent une requête après l'autre. Cependant, dans plusieurs cas, il y a un délai entre les requêtes d'une même application, et ainsi d'autres requêtes provenant d'autres applications peuvent arriver et interrompent ces premières requêtes. Dans ce cas, un ordonnancement "*work-conserving*" n'apporte pas de bonnes performances. Nous allons illustrer ce problème avec les différentes stratégies ci-après.

Ordonnancement visant à favoriser un ordre séquentiel

Ce type d'ordonnancement a pour but de favoriser la " sequentialité" dans les accès et réduire ainsi les déplacements sur le disque (cf. 1.1.4). Par exemple l'ordonnancement de type SPTF (*Shortest Positioning Time First*) exécute un ensemble des requêtes consécutives d'une même application avant de traiter les demandes provenant d'une autre application (figure 1.3). Cependant et d'un point de vue pratique, les requêtes de ces deux applications arrivent de manière synchrone (figure 1.4). Si l'ordonnancement est de type *work conserving*, il essaie de maximiser l'utilisation du disque : dès lors qu'une requête est terminée, une suivante est immédiatement traitée. S'il y a seulement une requête dans la queue, elle est sélectionnée. Dans ce cas, l'ordonnancement SPTF devient de type FCFS (*First Come First Served*) et les requêtes d'une même application ne sont pas exécutées de manière consécutive. La performance est donc fortement diminuée.

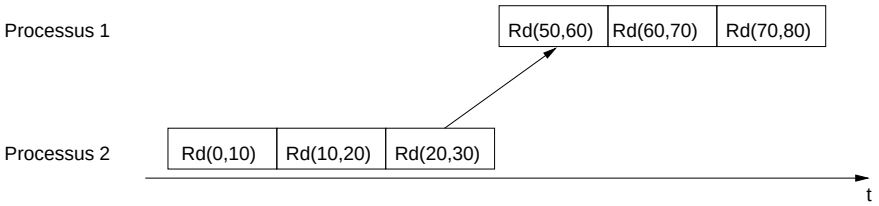


FIG. 1.3 – Ordonnancement pour réduire le temps de recherche sur disque - SPTF.
Rd(x,y) = Lire de l'offset x à y.

Ordonnanceur de partage proportionnel

Ce type d'ordonnancement partage le disque entre plusieurs applications selon une proportion définie. Par exemple, s'il y a deux processus et que la proportion est 1 :2 : dans un même temps, le nombre de requêtes exécutées de la première application correspond à la moitié de celui de la deuxième application

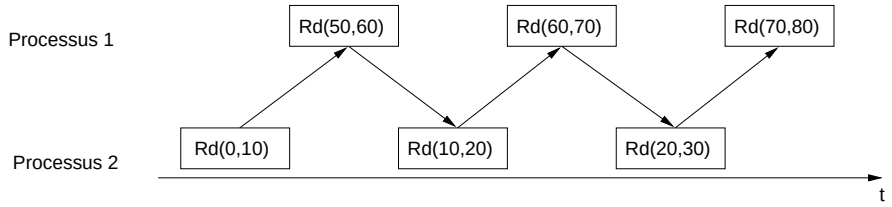


FIG. 1.4 – SPTF devient de type FCFS.

(voir figure 1.5). Cependant et une fois encore, si les requêtes arrivent de manière synchrone, un ordonnancement de type *work conserving* va engendrer une stratégie de type FCFS et la proportion devient 1 :1 (figure 1.4).

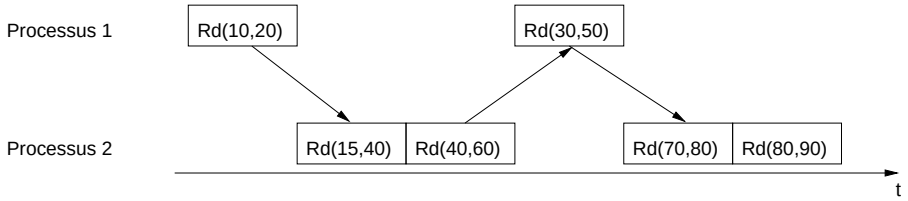


FIG. 1.5 – Ordonnanceur de partage proportionnel

Méthode d'ordonnancement anticipé

A partir des exemples ci-dessus, nous pouvons constater qu'une stratégie qui se révèle être intéressante en théorie peut devenir inapproprié en pratique. Les ordonnancements bas niveau souvent basés sur des approches *work conserving* en étant la principale cause. L'ordonnancement anticipé (*anticipatory scheduling*) [10] consiste à attendre un bref délai avant de choisir la prochaine requête. Chaque fois qu'une application finit d'exécuter une requête, l'ordonnancement ne choisit pas toute de suite une autre requête mais attend un bref délai dans "l'espoir" qu'il puisse recevoir d'autres demandes de la même application. Le temps d'attente est souvent très petit. Il est basée sur le coût d'un éventuel déplacement et est dépendant de la politique d'ordonnancement utilisée. Un ordonnancement de ce type est appelé *non work conserving*. Cette méthode a été intégrée au sein du noyau Linux pour les pilotes des disques.

Chapitre 2

Systèmes existants

Dans ce chapitre, nous abordons plusieurs systèmes de gestion de données : systèmes de fichiers distribués et/ou parallèles, librairies spécialisées dans les E/S parallèles, outils de transfert de données à l'échelle des grilles. Comme le système aIOLi (cf. 3.1), ces solutions se concentrent sur les critères de performances au sein des environnements parallèles. C'est pourquoi nous avons souhaité les présenter.

2.1 Systèmes de fichiers distribués/parallèles

Les systèmes de fichiers distribués permettent à plusieurs processus répartis ou non sur plusieurs machines d'accéder à un ensemble de données partagés. Les propriétés fondamentales qu'ils doivent garantir sont la transparence d'accès (les interfaces d'accès aux données doivent être les mêmes que celles employées pour les données locales), la tolérance aux pannes (aucune information ne doit être perdue), la réactivité (critère de performance) et enfin la scalabilité (passage à l'échelle).

Nous n'avons pas souhaité présenter de manière complète toutes les solutions disponibles ; plusieurs travaux s'étant concentrées à cette tâche [22, 23, 6]. Après un rapide rappel de certaines notions décrites dans [7], nous présenterons des systèmes ayant une relation avec notre travail.

2.1.1 Généralités

Les systèmes de fichiers distribués sont classés en deux catégories :

Les modèles à base de messages : Dans cette approche, il existe des machines (serveurs) qui possèdent et exportent les fichiers et des machines qui veulent monter et accéder à ces fichiers (clients). Il n'y pas de contrainte avec l'architecture sous-jacente. Le système de fichiers NFS (*Network File System*) repose sur ce modèle.

Modèle à accès direct : Dans le modèle à base de message, l'existance d'un serveur centralisé peut causer le problème de goulot d'étranglement. Dans le modèle à accès direct, les données sont distribuées dans le système. Plusieurs machines peuvent accéder en concurrence aux périphériques de stockages. Le système de fichiers GFS (*Global File System*) est un exemple de ce modèle.

Bien que les systèmes de fichiers distribués puissent gérer des accès "parallèles" provenant de plusieurs processus ; ils n'ont pas été conçus spécifiquement dans ce but. Par exemple dans le système NFS, un serveur centralisé gère la totalité du processus de gestion (mise à jour des meta-informations[1] ainsi que les transferts des données depuis et vers les processus clients). Ainsi, les accès parallèles sur un même fichier sont sérialisés et ce même quand ils concernent différentes parties de ce fichier. Les périphériques de type RAID ont permis de résoudre en partie ces problèmes mais restent des solutions onéreuses. S'appuyant sur ces concepts de répartition des données sur plusieurs unités de stockages, des nouveaux systèmes, communément appelés, systèmes de fichiers "parallèles" ont été proposées. Ces solutions reposent sur des caractéristiques communes [24] tel que :

[1] Les informations sur les fichiers : estampille de création/modification, taille, droit d'accès ...

Distribution de données : Les fichiers sont répartis sur plusieurs disques (“*data striping*”). Ainsi, plusieurs accès peuvent être traités simultanément. Les nœuds qui sont en charge de gérer les disques sont appelés des nœuds d'E/S, les autres nœuds étant des nœuds de calcul.

Utilisation des techniques “d'agrégation” : Plusieurs systèmes de fichiers parallèles ont intégrés des techniques similaires aux “*Collective I/O*” toujours dans le but de minimiser les petites requêtes et diminuer les temps de transfert.

Les systèmes de fichiers parallèles les plus connues sont PFS (“*Intel's Parallel File System*”), PIOFS (“*IBM's Parallel Input/Output File System*”), Galley, Vesta, ...et plus récemment PVFS (“*Parallel Virtual File System*”) version 1 et 2. Les derniers projets en cours proches de nos travaux sont les systèmes NFS version 4 et Clusterfile que nous présentons par la suite.

2.1.2 NFS Version 4

Système de fichier NFS

NFS (*Network File System*) est un système de fichiers permettant à plusieurs machines d'accéder aux disques distants de manière transparente, tel un disque local. NFS s'appuie sur un modèle client/serveur : une machine qui exporte son système de fichiers est appelée un serveur NFS, d'autres machines qui montent ce système de fichiers sont appelées des clients. NFS utilise le protocole RPC (*Remote Procedure Call*) et l'encodage des données XDR (*eXternal Data Representation*). La figure 2.1 décrit l'architecture générale de ce système.

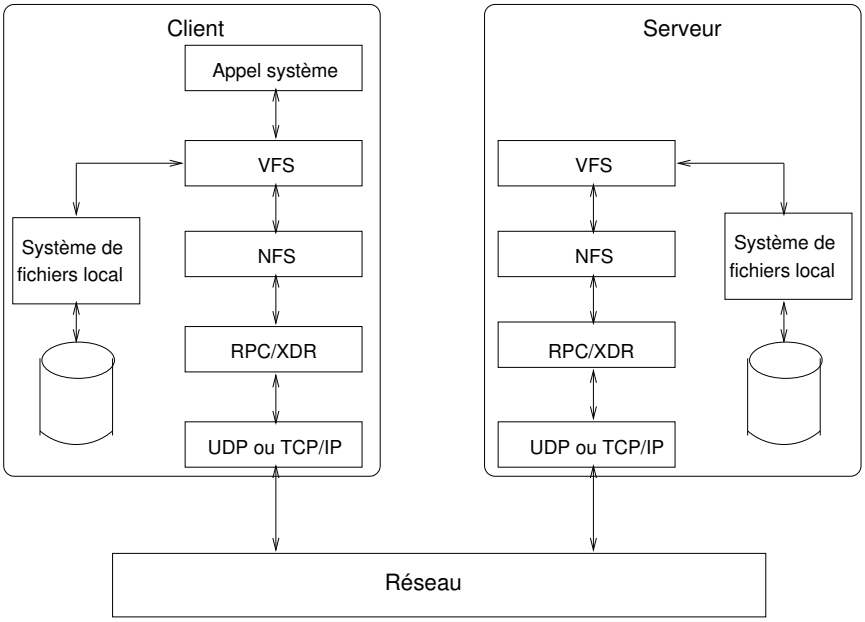


FIG. 2.1 – NFS

Depuis son apparition en 1985, NFS a beaucoup évolué. La dernière version proposée est la 4, [25]. Alors que les précédentes approches étaient axées vers des systèmes de partage de données au sein d'un réseau local, cette quatrième version s'attarde sur des contraintes liées aux grappes. Les changements principaux sont :

Réplication et migration de système de fichiers : NFS V4 permet à un système de fichiers de migrer ou de se dupliquer sur un autre serveur. Dans le cas d'une migration, un client interroge le serveur afin d'obtenir le nouveau système en charge du système de fichiers. Dans le cas d'une réplication, le client interroge le serveur afin d'obtenir une liste de tous les réplicats (attribute $fs\_locations$) afin d'accéder à la donnée correspondante.

Verouillage de fichier : Le changement principal dans le mécanisme de verrouillage dans NFS V4 est la notion de "*lease*" (ou contrat). Un *lease* est un temps borné attribué par le serveur à un client. Pendant ce temps le serveur ne fournit pas le contrôle conflictuel sur le même fichier à un autre client. Après le temps de *lease*, le contrôle est annulé si le client ne fait pas un avenant avec le serveur.

Cache client et mécanisme de délégation : Dans NFS V4, comme dans les anciennes versions, le client peut exploiter le mécanisme de cache aussi bien sur les méta-données (répertoires, attributes) que sur les données (contenu de fichiers) afin de réduire le nombre de requête et augmenter les performances globales. Les informations sur les méta-informations sont cachées pendant une durée de temps qui est déterminée par le client. Chaque fois qu'un client ouvre un fichier caché, il va demander au serveur si ce fichier a été modifié ou non.
Le serveur peut déléguer quelques responsabilités au client pendant un temps. Alors le client peut modifier les fichiers dans son cache sans contacter le serveur jusqu'à ce que le serveur notifie que d'autres clients veulent écrire dans ces fichiers. Une délégation peut être supprimée par le serveur si nécessaire. Ce mécanisme de délégation améliore l'utilisation de cache puisque les vérifications périodiques sur les états de fichiers, précédemment nécessaires, peuvent être évitées.

Serveur avec état (*"stateful"*) : NFS V4 n'est plus un protocole sans état tel qu'il était dans les versions anciennes. L'utilisation des mécanismes de verrouillage de fichiers et de délégation lui demande de maintenir des informations sur chaque client. Quand le client contacte le serveur la première fois, il doit fournir ses informations d'identification au serveur. Puis le serveur lui retourne un *clientid* (de 64 bits). Ce *clientid* est unique et il est utilisé pour identifier le client dans les session suivantes.

TCP est obligatoire : NFS V2 et NFS V3 permet d'utiliser UDP et TCP, mais NFS V4 n'utilise que TCP ("*stateful*").

Requêtes composées : NFS V4 permet d'utiliser les requêtes (ou opérations) de type *COMPOUND*. Une requête de ce type se compose de plusieurs autres requêtes. De manière similaire, une réponse peut être une agrégation de plusieurs réponses. Les requêtes *COMPOUND* réduisent le nombre des requêtes nécessaires pour les transactions.

pNFS : une extension de NFS V4

Bien que NFS v4 a ajouté plusieurs nouvelles caractéristiques, il reste un système *"single server"* dans lequel le serveur NFS est en charge de coordonner tous les accès des clients et de gérer le transfert de données. Le pNFS (*parallel NFS*) [26] est une extension de NFS V4 qui permet à plusieurs clients d'accéder directement aux différents systèmes de stockages. L'idée de pNFS est de séparer le flux de contrôle et le flux de données. L'avantage de pNFS est la répartition de la charge de travail entre le serveur, les clients et les systèmes de stockage. Dans ce modèle, il existe un serveur NFS unique qui est en charge de gérer le système de fichiers et les clients tandis que les transferts sont exécutées entre les clients et les systèmes de stockage (approche comparable au système PVFS, [21]).

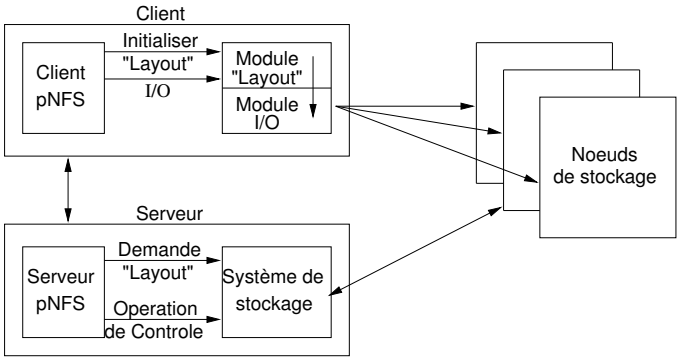


FIG. 2.2 – Architecture d'un système pNFS

L'architecture d'un système pNFS, illustrée dans la figure 2.2, se compose de plusieurs modules :

Module *"Layout"* : il connaît toutes les informations nécessaires pour accéder aux nœuds de stockage, c'est-à-dire la distribution physique des blocs de données de fichiers sur ces nœuds. D'abord le serveur pNFS doit transmettre au client une description de "*layout*" de ce système de stockage. Par la suite, cette information est transmise au "*layout driver*" en charge d'exprimer les requêtes de client pNFS pour les divers nœuds de stockage.

Module *I/O* : il est en charge d'exécuter les E/S à bas niveau.

Dans ce modèle, le serveur pNFS joue le rôle d'un serveur de méta-données. Il est en charge de gérer l'espace de nommage, permissions d'accès, partage et coordination des clients. Les nœuds de stockage sont en charge du transfert des données vers les clients.
Les opération sur le "*layout*" sont :

LAYOUTGET : obtenir les informations d'accès de fichier (*file layout*) de nœuds de stockage.

LAYOUTCOMMIT : Le client utilise cette commande pour confirmer les changements sur un fichier (par exemple après une commande d'écriture).

LAYOUTRETURN : Le client utilise cette commande pour informer au serveur que le *file layout* n'est plus nécessaire pour lui.

CB_LAYOUTRETURN : Le serveur retire les informations sur *file layout*

GETDEVINFO : Fonction *call back* du serveur pour retirer les information sur *file layout* du client.

Le système de fichier NFS V4 est en phase de stabilisation et devrait être rapidement remplacer la version 3 actuellement incluse dans le système LINUX.

2.1.3 Clusterfile

Clusterfile [27], [28], est un système de fichiers parallèle développé à l'université de Karlsruhe en Allemagne. Dans ce système, chaque fichier se compose de sous fichiers distribués sur un ou plusieurs nœud d'E/S (figure 2.3).

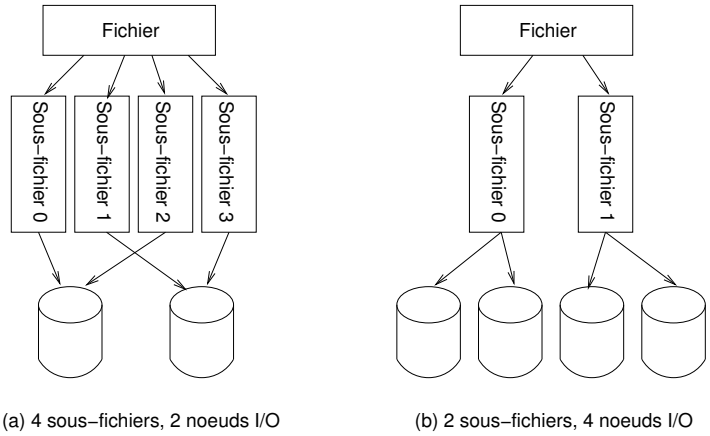


FIG. 2.3 – Distribution des données dans Clusterfile

Le système Clusterfile se compose de trois composants principaux :

Un composant de gestion de méta-données : Ce composant maintient à jour les informations concernant la structure des fichier ainsi que la distribution physique des sous fichiers sur les différents disques.

Plusieurs serveurs d'E/S : Les serveurs d'E/S ont en charge l'exécution des opérations de lecture et d'écriture sur les sous fichiers. Ces serveurs gardent également les informations sur les méta données des sous fichiers et les fournissent au composant de gestion de méta-données.

Une librairie d'E/S : Cette librairie est utilisé par les nœuds de calcul pour exécuter des opérations d'E/S sur le système de fichiers. Elle fournit une interface enrichie comprenant les routines standards Unix pour les utilisateurs.

Le système Clusterfile permet aux utilisateurs de spécifier leurs accès par des vues (“*View I/O*”, basée sur les vues utilisées au sein des SGBD). Une vue est une séquence d’adresse contiguë d’octets, elle est composée par une ou plusieurs régions d’un fichier. Les utilisateurs peuvent créer leurs vues en utilisant la commande `CLF_setview`. La vue alors créée, est envoyée au serveur d’E/S. Les serveurs d’E/S utilisent ces informations afin de prévoir les accès et donc d’optimiser les opérations d’E/S. Cette approche évite l’envoi de plusieurs requêtes indépendantes. De plus, un tel modèle d’accès global peut être utilisé pour l’ordonnancement des E/S ou l’amélioration des techniques de cache ou de préchargement. Plusieurs travaux sont encore en cours et tentent d’améliorer le système.

2.2 Librairies d’E/S parallèles

Le but des librairies d’E/S parallèles est de fournir aux utilisateurs des interfaces de programmation de haut niveau intégrant diverses techniques d’optimisations afin d’améliorer les performances et ce quelque soit le système de stockage sous jacent (critère de portabilité). Cependant, ces librairies sont souvent difficiles à utiliser et requièrent de la part des utilisateurs une profonde connaissance de l’ensemble de leurs subtilités afin d’en tirer le meilleur gain. Après plusieurs solutions et dans un souci d’uniformisation des interfaces, le consortium MPI a défini le standard MPI I/O [2] [29]. L’implantation la plus déployée de ce standard est la librairie ROMIO, [12] ; nous la décrivons en seconde partie de cette section. Parallèlement, plusieurs librairies précédemment proposées continuent d’être utilisées. La librairie Panda (“*Persistence AND Array*”) [30] a été développé pour traiter les problèmes d’E/S au sein des tableaux “multidimensionnels”. Nous avons choisi de la présenter car le modèle d’implantation est similaire au modèle aIOLi (cf. 3.1).

2.2.1 Panda

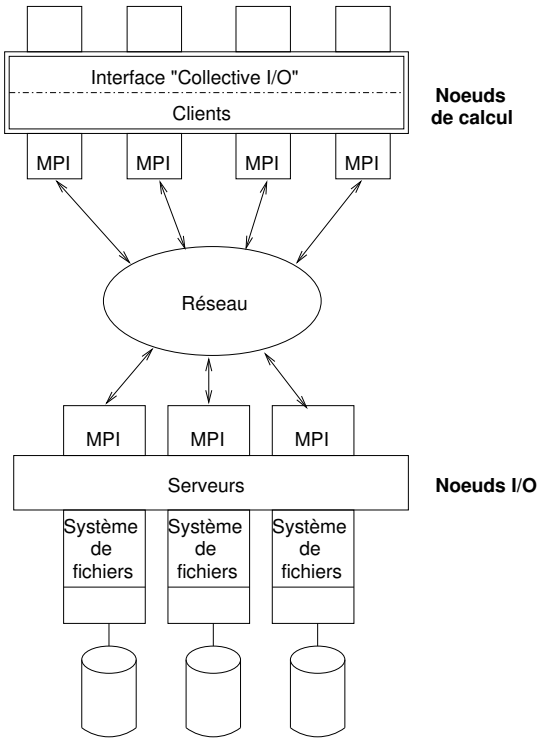


FIG. 2.4 – Architecture de Panda

[2] Disponible au sein des spécifications MPI 2 `http://www.mpi-forum.org/`.

Panda [30] est une librairie d'E/S pour la gestion des tableaux à plusieurs dimensions. Panda a été conçu dans le cadre d'applications parallèles s'exécutant sur plusieurs nœuds SMP. Un modèle de type "*Server-Directed I/O*" a été retenue (cf. 1.2.3).

Les techniques employées dans Panda pour optimiser les Entrées/Sorties sont :

Utilisation de technique *array chunking* permettant de traiter les tableaux de taille conséquente par sous blocs(*subarray chunks*) entre la mémoire et les disques. L'idée principale s'appuie sur le fait que le stockage linéaire d'un tableau n'est pas pratique lors de traitement en parallèle. La solution proposée ("*array chunking*") consiste à diviser un grand tableau en plusieurs morceaux qui pourront être maniés de manière indépendante et surtout en parallèle. Cette technique incluse déjà au sein des processus de gestion de la mémoire est exploitée ici pour le stockage sur disque.

Utilisation d'une interface de haute niveau permettant une gestion plus fine des sous-systèmes de stockages.

Amélioration de débit de disque grâce à l'approche collective ("*Server Directed I/O*"). L'ensemble des accès est sérialisé.

La sérialisation des accès, simple en théorique, reste complexe à mettre en œuvre. En effet, plusieurs systèmes d'E/S ne fournissent pas d'interfaces de haut niveau permettant de découvrir de manière dynamique les accès séquentiels. Par ailleurs, les requêtes arrivent souvent de manière désordonnée et il faut donc les réordonner. Le principe de " serialisation" des accès au sein de Panda est obtenu grâce à l'utilisation de routines spécifiques permettant d'exploiter plus finement les sous systèmes d'E/S.
La librairie se compose de deux composants (figure 2.4) : un module client (nœud de calcul) et un module serveur (nœud de stockage). L'application utilisateur est exécutée sur les nœuds de calcul et communique avec les modules clients par une interface de type "*Collective I/O*" spécifique aux tableaux multidimensions. Un client "master" et un serveur "master" sont sélectionnés parmi les clients et les serveurs correspondants pour être représentants de ces entités (le but étant de coordonner les actions de chaque côté et de réduire le nombre de messages transmis entre les clients et serveurs). Chaque fois qu'une application veut exécuter une requête collective, le client "master" envoie au serveur correspondant une description des schémas de données des tableaux en mémoire et sur le disque. Le serveur "master" diffuse alors cette information aux autres serveurs. La répartition des données aussi bien sur les disques qu'en mémoire étant connue par tous les serveurs, les requêtes d'E/S peuvent être exécutées de manière séquentielle en agrégeant les demandes consécutives. Une fois que l'exécution de cette requête collective est terminée, les serveurs membres informent le serveur "master". Ce serveur"master" va indiquer au client "master" que tout est fini. Le client "master" informe à son tour tous les autres clients.

Bien que l'architecture "*Server-Directed I/O*" soit une variante de "*Disk-Directed I/O*", il y a plusieurs différences entre eux. Dans la méthode "*Disk-Directed I/O*", les tableaux sont stockés selon l'ordre linéaire traditionnel et les blocs physiques de disques sont ordonnés pour obtenir de meilleures performances. Au contraire, dans la méthode "*Server-Directed I/O*" de Panda, les tableaux sont stockées dans les "*chunks*" et les optimisations se basent sur des accès séquentiels "logiques". Enfin, la méthode employée par Panda fournit un moyen supplémentaire de travailler sur les données par l'utilisation d'une API plus riche. Toutefois dans quelques cas, l'utilisation des serveurs d'E/S dédiés s'avère coûteuse. Le système a intégré une nouvelle approche appelée "*part-time I/O*". Dans cette méthode, un nœud peut jouer le rôle d'un client et d'un serveur I/O en permutation. Il peut être un serveur pendant un temps puis reprendre le rôle d'un client après avoir fini les E/S. Cette technique permet de réduire d'une part, le nombre des nœuds d'E/S dans le système. D'autre part, elle permet d'exploiter la totalité de l'espace disque disponible sur la grappe puisque chaque client peut jouer le rôle de serveur pour son propre disque. Néanmoins, elle requiert une mémoire plus grande sur les nœuds de calcul pour contenir les caches et les tampons nécessaires aux algorithmes d'E/S.

Le projet PANDA est toujours actif. Intégré au sein d'un projet de simulation spatial, la bibliothèque *RocPANDA* [31], se concentre sur les problématiques liées à la compatibilité des données et des accès entre les différents modules composant une application scientifique. La principale contrainte étant la permanente évolution de tous ces modules et qui de plus sont développés par plusieurs groupes de recherches souvent indépendant et programmant sous des langages et selon des règles différentes *C*, *C++*, *HPF*, ...

2.2.2 ROMIO

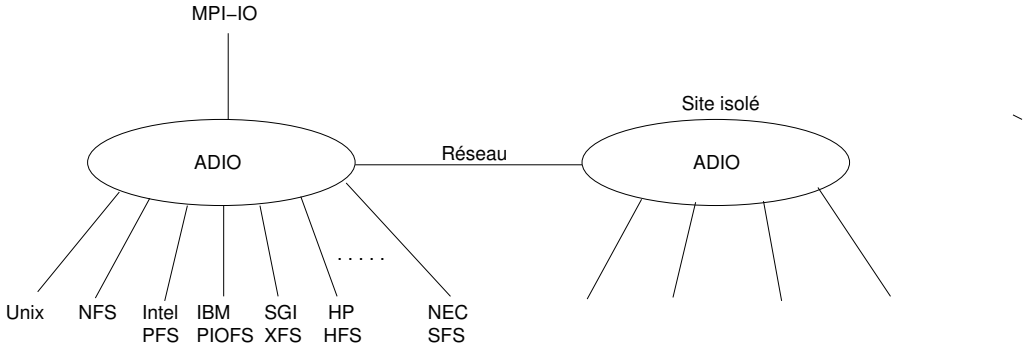


FIG. 2.5 – Architecture de ROMIO

Comme nous avons présenté dans 1.2, l'envoi de plusieurs petites requêtes peut fortement dégrader les performances. Le standard *MPI I/O* permet aux utilisateurs de spécifier des schémas d'accès ("*File view*") décrivant soit des "*simple* ou *nested strided*". Comme pour les vues de Clusterfile (cf. 2.1.3), ces informations sont utilisées par la suite lors des opérations d'E/S. Dans *MPI I/O*, toutes les fonctions d'E/S standards (`open/read/write /close/lseek`) sont remplacées par les routines d'accès équivalentes (`MPI_File_open, MPI_File_read, MPI_File_write, MPI_File_close, MPI_File_seek`) et les types MPI (*MPI datatypes*) permet de définir des vues complexes.

ROMIO [12] est l'implantation la plus utilisée du standard *MPI I/O*. Développé au laboratoire national d'Argonne, elle intègre deux types d'agrégations : le *Data Sieving* et l'approche collective *Two Phases* (cf. 1.2) ROMIO se scinde en 2 parties principales : l'"*ADIO*" (*Abstract Device interface for I/O*) est la couche assurant la portabilité du système. Elle se compose des fonctions basiques des E/S et offre ainsi la possibilité aux couches hautes d'être exécutées au dessus de plusieurs systèmes de fichiers comme NFS, PVFS, Unix, HP HFS, IBM PIOFS (aujourd'hui, l'ensemble des nouveaux systèmes de stockages offrent une interface ADIO). La seconde partie implémente les optimisations (*Data Sieving*, *Collective I/O*). Quatre types de routines d'accès sont disponibles :

Niveau 0 : requête indépendante, chaque processus exécute une commande indépendante (MPI_File_read) pour accéder à une région contiguë de données (surcharge simple de l'appel POSIX `read()`).

Niveau 1 : requêtes collectives, ce niveau est similaire au niveau 0 mais les processus utilisent une commande collective ($MPI\_File\_read\_all$) permettant aux processus participant d'appliquer une approche de type *Two Phase*.

Niveau 2 : *Data Sieving*, chaque processus crée un *datatype* pour décrire ses accès non contigus ($MPI\_File\_set\_view$) et exécute une seule commande indépendante (MPI_File_read) pour récupérer les données. La simple utilisation de la vue permet de basculer du niveau 0 à ce niveau.

Niveau 3 : *Two-Phase + File view*, chaque processus crée un *datatype* pour décrire ses accès non contigus ($MPI\_File\_set\_view$) et exécute une commande collective ($MPI\_File\_read\_all$) pour récupérer les données. L'approche *Two-Phase* va alors analyser toutes les vues et appliquer ensuite la répartition des accès.

Le schéma extrait de [32] illustre ces 4 niveaux.

2.3 Gestion et de transfert à l'échelle des grilles

Plusieurs applications déployées sur les grilles produisent grandes quantités de données (teraoctets ou petaoctets) et ces données sont souvent partagées entre plusieurs utilisateurs qui sont répartis sur plusieurs endroits (quelques fois au niveau intercontinental). Pour utiliser et gérer efficacement ces données, il nous faut des nouveaux outils. Plusieurs travaux tentent d'apporter des solutions axées autour des systèmes P2P

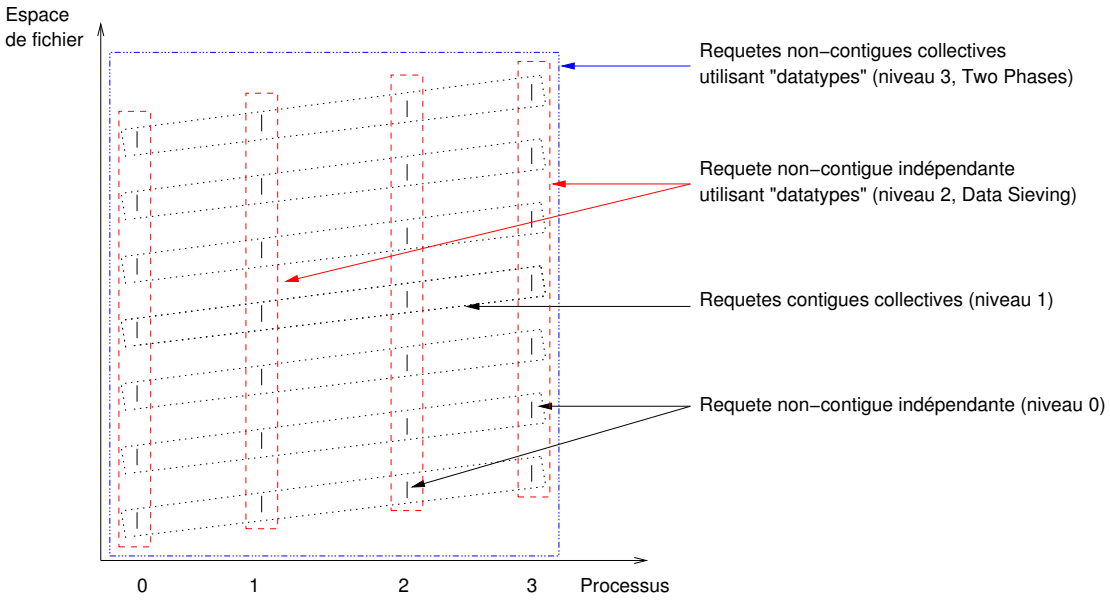


FIG. 2.6 – Niveaux d'optimisations dans ROMIO
A chaque niveau supérieur, les requêtes sont plus larges et donc plus efficaces

("pair à pair"). Toutefois, ils sont en cours de développement et non pas encore été totalement évalués. Le projet Globus [3] fournit un outil de transfert, GridFTP que nous allons présenter.

2.3.1 GridFTP : Protocole de transfert de données des grilles

Le protocole GridFTP est un extension du protocole FTP [33]. Il est utilisé comme un protocole standard pour transférer de données dans les environnements de grilles. Les extensions principales de ce protocole sont :

GSI : GridFTP support l'infrastructure GSI (*Grid Security Infrastructure*) dans laquelle l'authentification est basée sur le mécanisme de cryptographie à clé publique et le protocole SSL (*Secure Socket Layer*).

Contrôle de transfert de données serveur/serveur : Ce mécanisme permet à un client d'initier et de contrôler le processus de transfert de données entre deux serveurs de stockage.

Transfert de données en parallèle : GridFTP peut utiliser plusieurs flux TCP en parallèle pour augmenter la bande passante.

Transfert de données stripées : Les données d'un même fichier peuvent être réparties sur plusieurs serveurs de stockage. GridFTP permet d'utiliser plusieurs flux de données pour transférer données stripées en parallèle.

Transfert partiel de fichier : GridFTP permet de transférer une partie spécifique de fichier (au lieu de tout le fichier complète).

Négociation automatique de la taille des fenêtres TCP : GridFTP permet de négocier automatiquement et manuellement de la taille des fenêtres TCP.

Le mot GridFTP indique non seulement le protocole mais aussi une famille d'implantation et des outils de ce protocole. L'implantation de ce protocole repose sur deux librairies *globus_ftp_control* et *globus_ftp_client*. De plus, il y a un outil accessible en ligne de commande *globus-url-copy* similaire à la commande `scp` pour les utilisateurs de grilles. Le projet Globus fournit le service de transfert de fichier fiable RFT (*Reliable File Transfer*) pour contrôler le processus de transfert. Il supporte le transfert entre les serveurs GridFTP et FTP. Ce service utilise une base de données pour enregistrer l'état des transferts.

[3] www.globus.org

Si un transfert a échoué, il peut être reexécuté à partir de la dernière position. Le client peut soumettre une demande de transfert qui est alors prise en charge par ce service (exécution, surveillance, tolérance aux pannes). Le client peut questionner le service RFT sur l'état de transfert ou attendre le signal de complétion.

Deuxième partie

aIOLi

Chapitre 3

aIOLi : gestion des E/S parallèles

Le laboratoire ID-IMAG (Informatique et Distribution) est spécialisé dans la réalisation d'outils dédiés au calcul parallèle à haute performance et à leur validation sur des applications réelles. Une équipe du laboratoire s'est spécialisée dans l'étude et la mise en œuvre de techniques efficaces permettant le partage de ressources entre plusieurs applications parallèles au sein d'une grappe (mémoire partagée et système de fichiers principalement). En d'autres termes, il s'agit d'exploiter au mieux les caractéristiques du réseau d'interconnexion (débit des liaisons, existence ou non d'arêtes disjointes ...) et des caractéristiques de chaque nœud (capacité de stockage, puissance CPU, ...). C'est dans le cadre de ces recherches et autour de la collaboration LIPS[1] que s'inscrit le projet aIOLi.

Le système aIOLi est né du besoin de disposer de routines performantes pour l'accès aux données au sein d'une grappe sans pour autant exploiter une API complexe et fastidieuse. En s'appuyant sur cette idée et après avoir étudié de manière approfondie les critères et les comportements qui ont un impact sur les performances d'E/S (principalement l'envoi parallèlisé de requêtes, cf. chapitre 1), un modèle a été proposé. Il consiste à mettre en œuvre un système de coordination (ou de régulation) entre les requêtes de différents processus afin d'en synchroniser l'envoi et si possible de les agréger. Le premier prototype aIOLi [2] [34], [35] que nous allons présenter est une librairie efficace pour les accès de données au sein d'un nœud SMP (intra-nœud). Il permet de résoudre les problèmes ci-dessus de manière transparent avec les utilisateurs. Mon travail a consisté à distribuer ses concepts au niveau de la grappe (inter nœud).

3.1 Présentation du système aIOLi

La librairie se compose de deux composants : un module client qui surcharge les appels standards C (`open/close/read/write/lseek`) afin de les diriger vers un module serveur (démon aIOLi) en charge de traiter ces accès. Chaque fois qu'un processus génère une requête, le module client va envoyer cette requête au démon où elle est enregistrée dans une file d'attente avant d'être transmise au serveur de données.
Le démon est un processus "multithreadé" : un thread "receiver" reçoit les requêtes des clients et les enregistre dans les files d'attente correspondantes ; des threads I/O analysent ces files, agrègent les requêtes contiguës avant d'exécuter des appels systèmes réels. Le démon aIOLi a pour but de synchroniser l'envoi des requêtes afin qu'elles soient envoyées de manière séquentielle et éviter l'envoi parallélisé qui comme nous l'avons dit peut se révéler inefficace. Le stockage temporaire des requêtes au sein d'un point concentré nous permet d'analyser les accès afin de trouver des possibilités d'agrégation et d'appliquer différentes techniques d'optimisation.

Un des points forts de cette librairie est sa simplicité d'utilisation. Elle ne requiert pas de la part des utilisateurs la prise en main d'une nouvelle librairie qui peut se révéler parfois complexe. Elle surcharge

[1] Collaboration BULL/INRIA
[2] an Input Output Library

seulement les interfaces standards C ce qui rend l'approche portable sur toutes les architectures POSIX (la quasi-totalité des architectures informatiques).

Les expérimentations de ce premier prototype ont donné des résultats prometteurs en comparaison au performance fournit par les interfaces POSIX ou même ROMIO (cf. 2.2.2). Une analyse plus détaillée du premier prototype ainsi que les évaluations sont disponibles à l'url suivante : `http://aioli.imag.fr`. Dans la section suivante, nous allons présenter l'intérêt d'une distribution des concepts exploités par aIOLi.

3.2 Coordination entre plusieurs SMPs

Les résultats fort encourageants obtenus sur un nœud SMP nous ont amenés à élargir nos travaux à l'échelle des grappes. En effet, comme nous l'avons plusieurs fois indiqué, l'envoi parallélisé de plusieurs requêtes sur un même serveur de données peut dégrader considérablement les performances. Une gestion contrôlée au niveau du client SMP mêlant régulation et agrégation a nettement améliorer les temps de traitements. Ainsi dans un système qui se compose de plusieurs nœuds (SMP), nous souhaiterions étudier les difficultés de mise en œuvre de techniques similaire afin d'en évaluer dans un second temps les éventuels profits.
Afin de faciliter cette mise en œuvre, nous allons tout d'abord analyser les contraintes imposées par une coordination inter-nœuds [3] permettant de "combiner" au mieux l'ensemble des requêtes à destination d'un même serveur : les principales difficultés se situent d'une part au niveau de la synchronisation des envois à destination d'un même serveur afin d'en maximiser l'utilisation (cf. 3.3) et d'autre part au choix d'une combinaison judicieuse des demandes afin d'obtenir les meilleures performances. Nous nous proposons d'évaluer plusieurs politiques d'ordonnancement susceptibles d'améliorer le temps de réponse des requêtes de manière globale (cf. 3.4).

Nous avons choisi de centraliser les requêtes au sein d'un nœud maître[4] aIOLi capable de proposer l'ordonnancement adéquat. La mise en place de point de décision centralisé a été approuvé dans les travaux précédents. Dans un premier temps, nous simplifierons le problème en utilisant des serveurs de données centralisés (de type NFS). Nous reviendrons par la suite, aux contraintes liées au système de stockage parallèle où les requêtes sont réparties sur plusieurs serveurs surchargés de manière hétérogène. Dans ce cas nous pourrons attribuer un master par serveur I/O ou intégrer dans chaque master la notion de répartition de données (*data striping*).

D'un point de vue globale, les requêtes provenant des nœuds membres d'un groupe sont transmises au nœud master où elle sont analysées afin de trouver des possibilités d'agrégation. Durant une seconde phase, une politique d'ordonnancement est appliquée. Concrètement, le but d'une telle solution logicielle est de pouvoir intervenir à 3 niveaux au sein de la gestion des E/S dans une grappe :

1. Coordination intra-nœud : agrégation et ordonnancement au sein d'un nœud pour traiter des accès I/O d'une même application. Par exemple dans une application de calcul d'une matrice, chaque élément de la matrice est calculé par une instance SPMD [5](tâche MPI). Les requêtes des différentes tâches sont agrégées et ordonnées par dépendance d'offset. C'est ce premier niveau d'optimisation que propose d'ores et déjà la librairie aIOLi.
2. Coordination multi-nœuds : proposer une synchronisation ainsi que des politiques d'ordonnancement au sein d'une même application mais sur plusieurs nœuds SMP. Par exemple, toujours dans l'application de multiplication de matrices, chaque nœud est en charge d'une partie de la matrice, cette sous partie également distribuée sur l'ensemble des processeurs du noeud. Dans ce cas, il est primordiale de synchroniser les requêtes afin de maximiser l'utilisation de serveurs de stockages tout en tenant compte du problème d'exclusion mutuelle présenté dans 3.3.
3. Coordination multi-applicatifs : combiner (1) et (2) en tenant compte de la concurrence entre les applications. Dans ce mode, plusieurs applications peuvent envoyer leurs requêtes dans le temps.

[3] Comme nous l'avons indiqué, la librairie aIOLi réalise à ce jour uniquement une coordination intra-nœud.
[4] nous utiliserons l'anglisiscme "master" par la suite.
[5] *Single Program Multiple Data*

Par exemple une application réalise une multiplication de matrices tandis qu'une autre effectue de la FFT. Dans une première phase, le processus d'agrégation établi un premier ordre parmi les requêtes (dépendance par offset). Cette agrégation "virtuelle" permet de sérialiser les accès afin de minimiser les déplacements et d'optimiser les politiques de cache (principalement "*read ahead*"). Des algorithmes d'ordonnancement sont alors appliqués afin de maximiser les performances tout en tenant compte des contraintes d'efficacité et d'équité.
Remarque : Une agrégation physique serait plus efficace mais imposerait de nombreuses contraintes (gestion de la cohérence de données, réplication, système de "*callbacks*" pour invalider les caches, ...). Nous avons donc préféré scinder le travail en deux afin de bien se concentrer sur chacune des implantations. La réalisation de cette seconde partie est d'ailleurs une éventuelle perspective de ce travail.

3.2.1 Architectures logicielles

Dans cette partie, nous nous attardons sur les topologies possibles de notre modèle.

Un master par serveur d'E/S

Dans cette configuration (figure 3.1), chaque nœud master est en charge de gérer les requêtes d'un même serveur I/O. Dans ce cas particulier, si le master est également le serveur d'E/S, il peut exploiter ses connaissances sur la distribution physique de données et est donc capable de proposer une stratégie similaire à la solution *disk-directed I/O* (cf. 1.2.3). De plus, la mise en place d'un tel service de régulation va cependant augmenter les problèmes de surcharge le cas échéant. Lorsqu'un serveur d'E/S est déjà fortement sollicité par le transfert de données, la mise en place d'un service plus complexe d'ordonnancement va accroître les problèmes de saturations. Il est donc nécessaire de prévoir une autre approche afin de répondre au critère de scalabilité. Toutefois, C'est cette approche (convenable au niveau de grappes) que nous avons utilisée lors de nos évaluations.

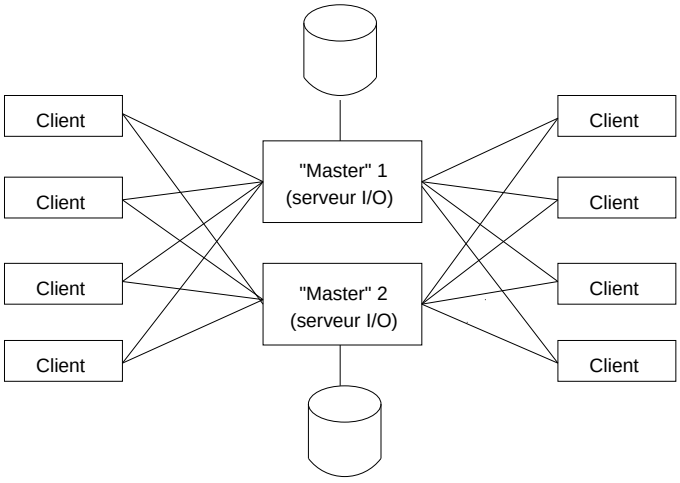


FIG. 3.1 – Un master par serveur d'E/S

Un master par sous groupe de nœuds

Dans cette approche, chaque nœud client appartient à un sous groupe qui est géré par un nœud master. Toutes les requêtes des membres de ce sous groupe sont envoyées au nœud master correspondant afin de les traiter. Un client doit connaître seulement le serveur qui est en charge de le gérer directement. Cette approche nous permet d'élargir facilement le système en utilisant un modèle hiérarchique. Dans ce modèle, les masters qui gèrent directement les clients sont appelés les masters de niveau 1. Un groupe de masters de

niveau 1 peut être géré par un master de niveau 2. Dans ce cas, toutes les requêtes des clients d'un master de niveau 1 sont considérées comme ses "propres" requêtes. Ce master va faire une première agrégation et sélection pour choisir une requête et l'envoyer au nœud master de niveau 2 (qui le gère). Maintenant, avec un master de niveau 2, tous les requêtes des masters de niveau 1 qu'il gère sont considérées comme ses requêtes. De manière similaire, plusieurs masters de niveau k sont gérés par un master de niveau k+1. Le nombre de niveaux est dépendant de la taille du système. Cette approche semble approprié aux grilles puisqu'elle permet d'élargir facilement le système. La figure 3.2 illustre cette approche.

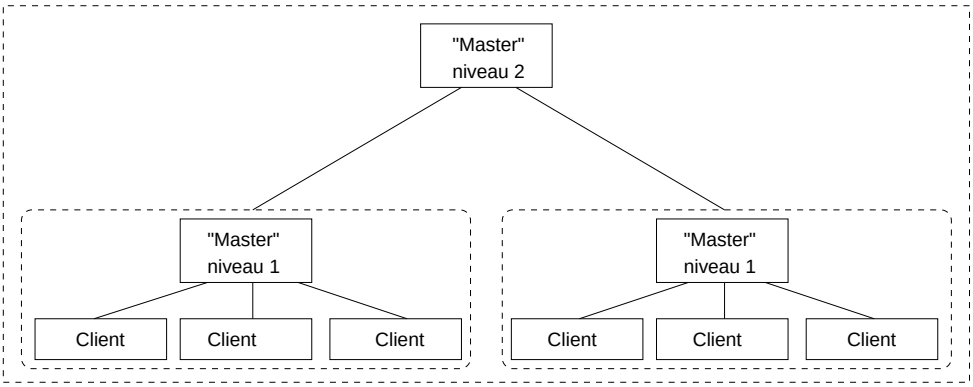
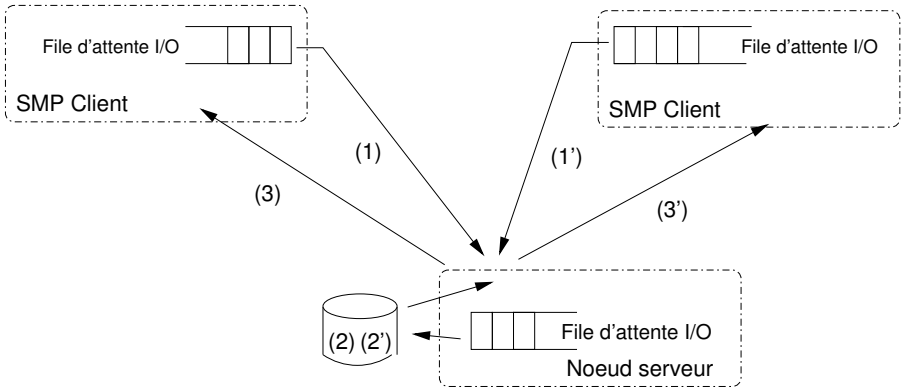


FIG. 3.2 – Un master par sous-groupe

3.3 Synchronisation des requêtes

Comme nous l'avons indiqué en 3.2, la première difficulté consiste à pouvoir "réguler" l'envoi de requêtes entre l'ensemble des nœuds SMP afin qu'une seule demande ne soit traitée par le serveur de stockage au même moment et ce, sans pour autant en diminuer son utilisation.



(1) et (1') Une requête est transmise au système de fichiers.
(2) et (2') Elle est exécutée sur le périphérique de stockage rattaché.
(3) et (3') La réponse est renvoyée au client.

Afin de maximiser les performances, nous devons réguler l'arrivée
des requetes sur le serveur tout en maximisant son utilisation.
Il faut donc éviter les conflits entre accès néfaste comme l'ont montré
les précédents travaux sans pour autant perdre un délai trop important
entre deux accès => Problème d'exclusion mutuelle distribuée

FIG. 3.3 – Les accès vers un système de stockage distant

Cette contrainte peut facilement être ramenée à un problème de type exclusion mutuelle distribuée. En

effet, plusieurs sites (dans notre cas les nœuds SMPs) veulent accéder à la même ressource partagée (le serveur de données). C'est ce problème que nous abordons dans cette partie.

3.3.1 Exclusion mutuelle distribuée

Le problème d'exclusion mutuelle a été étudié depuis fort longtemps, surtout dans les recherches concernant la synchronisation de processus. Plusieurs méthodes permettant de résoudre ce problème sont abordées dans [19]. Dans un système qui se compose d'une seule machine, le problème d'exclusion mutuelle peut être résolu facilement via l'utilisation de variables partagées (sémaphores). Cependant, dans les systèmes répartis, le problème devient plus compliqué puisqu'il n'y a pas d'accès direct à une mémoire globale ni à une horloge globale. Un excellent panorama des algorithmes utilisés dans ce type d'architecture est établi dans [36]. Selon les auteurs, les conditions qu'un algorithme d'exclusion mutuelle doit satisfaire sont :

Sans "deadlock" : Deux ou plusieurs sites ne devraient pas attendre un message (ou événement) qui n'arrivera jamais.

Sans famine : Tous les sites doivent avoir l'opportunité d'accéder à la ressource dans un temps fini.

Equité : Cette propriété implique que les requêtes devraient être exécutées selon leur l'ordre d'arrivée.

Tolérance aux fautes : Un algorithme d'exclusion mutuelle doit pouvoir continuer à fonctionner après une panne.

Les critères utilisés pour évaluer un tel algorithme sont :

Nombre de messages : qui sont utilisés dans l'algorithme pour chaque tour.

Délai de synchronisation : c'est le temps entre l'évenement "un site libère la ressource" et l'évenement "un autre site a accédé à cette même ressource". Un délai de synchronisation important entraîne une utilisation non optimale de la ressource, figure 3.4.

Temps de réponse : c'est le temps entre l'envoi par un site d'un message de demande d'accès à la ressource et le moment où il finit son travail sur la ressource.

Débit de système : c'est le ratio d'exécution de requêtes sur la ressource partagée. Si ds est le délai de synchronisation et E est le temps d'exécution moyen sur la ressource, alors ce ratio est définit suivant :

$$D = \frac{1}{ds + E} \tag{3.1}$$

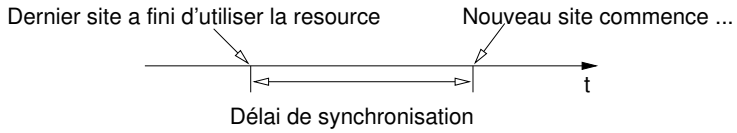


FIG. 3.4 – Délai de synchronisation

3.3.2 Les diverses solutions

Approche "simple"

Une approche simple (voir figure 3.5) pour le problème d'exclusion mutuelle distribuée est d'utiliser un site de contrôle pour synchroniser les requêtes. Chaque fois qu'un site veut accéder à la ressource, il envoie un message de demande au site de contrôle, le site de contrôle garde cette requête dans une queue et informe le site de demande quand la ressource est libérée. Bien que cette méthode soit assez simple à implanter, elle comprend deux inconvénients majeurs : le site de contrôle peut être surchargé et n'est pas

tolérant aux pannes[6], le délai de synchronisation est dépendant de la latence de l'architecture sous-jacente (temps requis pour l'envoi de 2 messages).

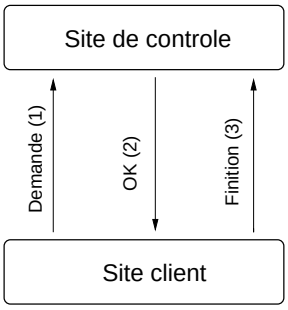


FIG. 3.5 – Algorithme simple pour le problème d'exclusion mutuelle

Les algorithmes à jeton

Dans ce type d'algorithme, il y a un jeton unique qui est partagé entre tous les sites. Un site peut accéder à la ressource seulement s'il possède le jeton. Dans ces algorithmes, la principale mission est de gérer le mouvement de jeton. L'algorithme le plus simple est de laisser le jeton parcourir tous les sites selon un ordre fixe. Cette méthode se révèle être très inefficace en ce qui concerne le critère de délai de synchronisation ($T * (N - 1)$ dans le pire cas, T est le temps nécessaire de transmission d'un message entre deux sites, N est le nombre de sites). Il existe plusieurs autres algorithmes de ce type comme celui proposé par Suzuki-Kazami, [37] avec N messages et un délai de synchronisation moyen de T ou encore l'algorithme de Raymond, [38], avec $log(N)$ messages et un délai de synchronisation moyen de $T * log(N)/2$.

Les autres algorithmes

Un des premiers algorithmes pour résoudre le problème d'exclusion mutuelle distribuée est abordé dans l'article classique [39] de Leslie Lamport. Cet algorithme requiert un délai de synchronisation T mais il utilise $3(N - 1)$ messages pour chaque tour de synchronisation : chaque fois qu'un site veut utiliser la ressource, il demande la permission de tous les autres sites. Un site qui ne veut pas utiliser la ressource va envoyer sa permission ; à l'inverse une priorité sera établie entre deux sites concurrents. Cette priorité est basée sur l'estampillage des requêtes en utilisant des horloges logiques. Par la suite, Ricart et Agrawala [40] nous donnent une optimisation de cet algorithme qui utilise $2(N - 1)$ messages. Plusieurs autres algorithmes basés sur l'utilisation des horloges logiques ont été suggérés ([41], [36]). Cependant tous ces algorithmes doivent utiliser un grand nombre de messages et sont donc comme la première solution fortement dépendant des performances réseaux.

3.3.3 Solution retenue

Parmi les diverses solutions existantes, la plus efficace selon le critère de nombre de message consiste en la mise en place d'un site de contrôle. Un des désavantages de cette approche, comme nous l'avons précédemment dit, est son fort délai de synchronisation (de $2T$). D'autres méthodes peuvent diminuer ce délai jusqu'à T (algorithmes de Suzuki-Kazami, de Lamport, de Ricart ...) mais il faut utiliser une redondance de messages. Dans notre contexte, notre but est de diminuer ce délai jusqu'à zéro (dans le cas optimum) tout en limitant le nombre de messages nécessaire pour réaliser la coordination. Pour parvenir à cela, nous allons nous appuyer sur une méthode de "prédiction" du temps d'exécution de la requête en cours. Cette méthode est décrit brièvement ci-après.
Côté client :

[6] *Single Point Of Failure.*

Chaque fois qu'un client souhaite accéder au serveur de stockage, il envoie un message `DEMANDE` au site de contrôle (dans notre cas aIOLi *master*) et attend jusqu'au moment où il reçoit un réponse `PREPARATION` contenant un délai préalablement calculé.

Le client reste en attente jusqu' à la fin de la période estimée avant de débuter à son tour le transfert de données.

Quand le client a fini son transfert, il envoie un message FINI au site de contrôle.

Côté site de contrôle / aIOLi *master* :

Quand le site de contrôle reçoit un message `DEMANDE`, il insère cette requête dans une queue.

Quand le site de contrôle reçoit un message FINI, il sait qu'un client a fini son travail et qu'un autre client a commencé à exécuter sa requête. Il va estimer le temps de transfert de ce client et puis envoyer un message `PREPARATION` au premier client dans sa queue. C'est cette information que le client utilise pour envoyer réellement sa demande en temps voulu au serveur. Le temps de transfert est estimé par la formule suivante :

$$temps\ de\ transfert = \frac{taille\ de\ requete}{debit\ de\ disque} \tag{3.2}$$

Nous pouvons constater que dans le cas optimum, le délai de synchronisation est 0, et le nombre de message pour chaque tour de synchronisation est 3 (un message DEMANDE, un message PREPARATION et un message FINI).

Malheureusement, le succès de cette méthode est liée en plus des caractéristiques réseau, à la justesse de la prédiction de temps de transfert de données. Dans le cas optimum, le temps réel est égal au temps prévu, figure 3.6 a. Cependant, si le temps réel est supérieur au temps prévu, le deuxième site va commencer son transfert alors que le premier site est en train d'exécuter sa requête, figure 3.6 b (turpidité des accès) ; enfin dans le dernier cas si le temps réel est inférieur au temps prévu, le disque n'est pas utilisé de manière efficace, figure 3.6 c).

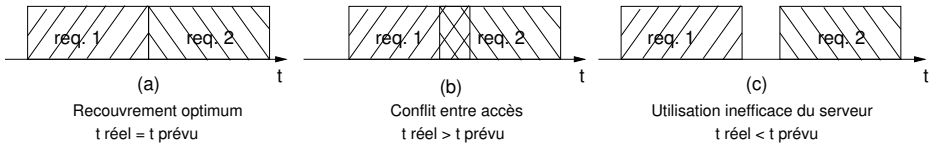


FIG. 3.6 – Synchronisation des accès

Pour résoudre le problème de recouvrement entre les accès, nous pouvons multiplier le temps de transfert prévu avec un nombre k (fixé selon les caractéristiques matérielles) pour garantir que le temps réel est toujours inférieur ou égal au temps prévu. Cependant, cette approche peut mener la situation où le délai de synchronisation est supérieur à T. Ainsi à chaque fois qu'un client finit son transfert, il va comparer son temps d'exécution réel avec le temps prévu :

Si $(treel - tprevu) < T$, il ne fait rien (dans ce cas le délai de synchronisation est supérieur à zéro mais inférieur à T)

Si $(treel - tprevu) > T$, il va envoyer un message `SUPPLEMENTAIRE` au site qui est en train d'attendre pour lui dire de commencer immédiatement son transfert.

L'avantage de cette approche est que, dans le pire cas, le délai de synchronisation est T et le nombre de message nécessaire pour un tour est 4 (`DEMANDE, PREPARATION, SUPPLEMENTAIRE, FINI`) et dans le cas optimum, le délai de synchronisation est 0 et le nombre de message est 3.

Par ailleurs et toujours dans un souci d'obtenir une prédiction la plus probable, nous nous proposons d'appliquer différentes méthodes comme celles exploitées dans [42]. Dans cette article l'auteur présente plusieurs méthodes pour la prédiction des paramètres dans un réseau (latence, débit de réseau ...) en utilisant des valeurs historiques. Ces méthodes peuvent être appliquées pour la prédiction du débit de disque. Les méthodes que nous avons l'intention d'utiliser sont décrites brièvement ci-après :

Méthodes basées sur la valeur moyenne (*mean-based methods*) : Ces méthodes utilisent la valeur moyenne des valeurs historiques comme la valeur de prédiction.

Méthodes basées sur la valeur médiane (*median-based methods*) : Ces méthodes utilisent un estimation médiane pour la valeur de prédiction.

L'auteur a aussi présenté une approche de sélection dynamique de ces méthodes. Selon cette approche, au moment de décision, une méthode sera sélectionnée si elle génère l'erreur la plus petite (à ce moment) par rapport aux autres méthodes.

3.4 Ordonnancement de requêtes

Les premiers travaux autour du prototype aIOLi ont eu pour but de valider l'intérêt des concepts d'agrégation et/ou de ré-ordonnancement au sein d'une même application *HPC*. S'il est possible d'envisager qu'une seule et même application puisse être attribuée à un nœud SMP de 2 ou 4 voies et que par conséquent l'ensemble des Entrées/Sorties disques distantes concernes les mêmes fichiers, il est impossible de reexploiter cette hypothèse au sein d'un entité SMP à plus forte échelle (16, 32 ou 64 processeurs) ou notamment au sein d'une grappe où plusieurs applications scientifiques sont menées à être exécutées sur la même période et donc à partager les mêmes ressources de stockages. Pour illustrer ces propos de ce que nous appelons le critère multi-applicatifs [7], nous allons nous intéresser à l'exemple suivant.
Supposons qu'il y a trois requêtes (numérotées 1, 2, 3) de trois différentes applications qui doivent être envoyées au même serveur de données. Les tailles de données de ces requêtes sont 10, 1, 5 Mo respectivement. Supposons que le temps nécessaire pour exécuter une requête est correspondant avec la taille de données de cette requête. Alors le temps pour exécuter ces requêtes sont respectivement 10, 1 et 5 unités de temps.
Par ailleurs, nous avons 3 ! (ou 6) possibilités d'ordonnancement : si l'ordre retenue pour exécuter ces trois requêtes est le "FIFO" (*First In First Out*), la première requête est exécutée immédiatement, la deuxième commencera 10 unités de temps plus tard et terminera son travail 1 unité après (soit un temps de réponse pour la deuxième requête de 1 + 10 = 11 unités). De manière similaire,la troisième requête aura un temps de réponse de 5 + 10 + 1 = 16 unités. Ainsi, la somme des temps de réponse pour ces trois requêtes est de 37 unités de temps (10 + 11 + 16).
Maintenant recommençons avec un ordonnancement différent : les requêtes sont exécutées selon l'ordre "SJF" (*Shortest Job First*). C'est-à-dire que la requête ayant la taille la plus petite est exécutée en priorité. Selon cette loi, l'ordre d'exécution de ces requête est 2, 3, 1. Le temps nécessaire pour exécuter la requête numéro 2 est 1 unité, Le temps de réponse de la deuxième requête est de 6 unités (1 + 5). Le temps de réponse de la troisième requête est de 16 unités (1 + 5 + 10). Dans ce cas, la totalité des temps de réponse pour les trois requêtes est cette fois ci de 23 (1 + 6 + 16).
Ainsi, chaque stratégie d'ordonnancement s'attarde sur un critère particulier de performances. Par exemple l'approche "FIFO" dans notre exemple minimise le temps de réponse maximum des applications. A l'inverse l'approche "SJF" est une bonne méthode pour minimiser la totalité des temps de réponse (cf. 1.3). Par conséquence, il est primordiale pour un problème donné, d'identifier le critère d'évaluation ou la fonction objective que nous souhaitons optimiser.

3.4.1 Choix du critère dans aIOLi

Le choix du critère dans notre modèle est complexe. En effet, nous souhaitons maximiser les performances en agrégeant/combinant les requêtes en de plus larges, tout en assurant un minimum d'équité entre les applications et éviter de manière rigoureuse les processus de famine. Les requêtes arrivent de manière indépendante tout au long de l'exécution d'une application ; il est d'ailleurs impossible d'en prévoir le nombre : mode *online*. Par ailleurs, des requêtes sur un même fichier peuvent être regroupées lors du processus "d'agrégation virtuelle" sur le master (lors d'une lecture par exemple si elles sont contigues), il est donc impossible de prévoir à tout moment la durée effective d'une requête : modèle non-clairvoyant. Nous avons choisi de mettre en œuvre une politique d'ordonnancement permettant de minimiser la totalité

[7] Différentes requêtes émanant de différentes applications.

des temps de réponse en garantissant par l'utilisation de seuils fixés l'équité entre les accès vers des ressources différentes (et donc potentiellement entre les applications, chacune des applications accédant dans la majeure partie des cas à des fichiers spécifiques).

3.4.2 Algorithmes d'ordonnancement proposés

Une adaptation de l'algorithme Weighted SJF

Comme nous l'avons présenté, l'algorithme SJF (cf. 1.3) est un algorithme efficace afin de minimiser la totalité de temps d'exécution. Cependant, c'est un algorithme *clairvoyant* qui peut causer des problèmes de famine sur des requêtes de taille importante. Dans [43] les auteurs ont présenté un algorithme appelé *"Weighted Shortest Time First"*. L'idée principale de cet algorithme est d'exploiter les avantages de l'algorithme SJF en réduisant le problème de famine. Pour cela, ils utilisent une mesure virtuelle du temps d'exécution de chaque tâche. Supposons que le temps d'exécution réel d'un travail est Tr, que le temps passé depuis l'événement d'arrivée de ce travail est E et que M est le temps d'attente maximum pour une requête ; alors le temps d'exécution virtuelle de ce travail est :

$$Tv = Tr * \frac{M - E}{M} \tag{3.3}$$

Chaque fois que la stratégie d'ordonnancement doit choisir une requête, il sélectionne la requête ayant la taille virtuelle la plus petite. Cette dernière, diminuant avec le temps, l'algorithme garantit qu'une requête sera exécutée bien que sa taille physique soit grande.

Néanmoins et dans notre modèle, la taille d'une requête n'est pas fixée d'avance : elle varie selon l'arrivée des différentes demandes dans le temps et le processus d'agrégation. Le nombre et le temps d'arrivée de ces requêtes n'étant pas connues d'avances, nous proposons de modifier la formule précédente par la suivante :

$$Tv = \sum_{i=1}^{i=n} Tvi \tag{3.4}$$

Dans cette formule, n est le nombre de requêtes, Tvi est le temps d'exécution virtuelle de la i^{ieme} requête qui est calculée selon la formule 3.3.

Même si cette approche réalise une pondération entre les dates d'arrivées et la taille de chacune des requêtes au sein d'une demande agrégée, l'algorithme comme tel peut encore engendrer des problèmes de famine. En effet, si la taille d'une requête "agrégée" croît de manière proportionnelle à la diminution de sa taille virtuelle, la famine réapparaît si en parallèle d'autres applications génèrent des demandes de tailles moindre. Pour résoudre ce problème, nous limitons la taille d'une requête "agrégée" (correspondant à une agrégation de plusieurs requêtes contiguës) par un seuil. Dans le cas ou le processus d'agrégation trouve une requête de taille supérieur à M, il retournerait 2 requêtes "agrégées" : une première de taille M et une seconde de taille $(Tr - M)$ avec une pondération sur la première requête lui permettant d'être prioritaire par rapport à la seconde. Nous choisissons de limiter la taille réelle au lieu de la taille virtuelle d'une requête composée puisque si nous limitons sa taille virtuelle, d'autres requêtes peuvent arriver dans le temps et être agrégées avec cette requête ; ainsi sa taille virtuelle diminue faiblement (elle est toujours inférieur à M) alors que sa taille physique augmente considérablement. L'algorithme WSJF est illustré dans l'exemple suivant (figure 3.7) :

A l'étape 1, il y a 4 requêtes de l'application A1, 3 requêtes de l'application A2. Supposons que le temps d'exécution de chaque requête est de 5 unités de temps et que M est égal à 30 ; après le processus de pré-traitement pour agréger les requêtes, la formule (2) nous donne la taille virtuelle des requêtes A1 (=20) et A2 (=15). Par conséquence A2 est exécuté avant.

A l'étape 2, quand la requête composée A2 se termine, une nouvelle requête A1 et 4 requêtes A3 sont arrivées. Ces requêtes sont alors réordonnés par le processus d'agrégation. Le temps d'attente de 4 premières requêtes A1 est de 15 unités de temps (c'est le temps d'exécution de 3 requêtes A2 sélectionnées lors de l'étape 1) et le temps d'attente de la nouvelle requête A1 est 0 ; par conséquent la taille virtuelle des requêtes A1 est de $4 * 5 * (30 - 15)/30 + 5 * (30 - 0)/30 = 15$. De même, la

taille virtuelle des requêtes A3 est de $4 * 5 = 20$. Les requêtes A1 sont prioritaires sur les requêtes A3 malgré que la taille réelle de la requête composée A1 soit supérieure à celle de requête composée A3.

A l'étape 3, pendant que les requêtes A1 sont exécutées, d'autres requêtes A3 arrivent. Toutefois, le processus d'agrégation ne retourne pas une unique requête agrégée mais deux (la taille de la requête composée dépasse. le seuil M). Dans ce cas, la première requête composée est prioritaire sur la deuxième (si nous appliquons la loi SJF, la deuxième est prioritaire puisque sa taille virtuelle est plus petite mais cet ordre là va pertuber la séquentialité et donc dégrader la performance de techniques de cache).

FIG. 3.7 – Algorithme WSJF

Une variante de MLF

Dans cette section, nous proposons un algorithme basé sur les concepts de l'algorithme MLF (1.3.1). Bien que les algorithmes SETF et MLF sont les algorithmes standards pour les modèles non-clairvoyant,

ils peuvent dans certains cas causer le problème de la famine [15]. La variante que nous proposons permet d'éliminer les problèmes de famine dans notre cas :

Chaque fois que la stratégie d'ordonnancement veut choisir une requête à exécuter dans la queue, il va proposer un quantum de temps d'exécution à chaque demande. Si une requête peut être exécutée dans ce temps, elle sera choisie ; si plusieurs requêtes satisfont cette condition, la décision sera donnée selon l'ordre FIFO.

Le quantum de temps n'est pas identique pour toutes les requêtes. Les nouvelles requêtes reçoivent des plus petites quanta que les anciennes requêtes. Si une requête n'est pas choisie après un tour de sélection ; dans le tour suivant, elle se voit proposer un quantum plus grand (k fois plus grand que le précedent). L'objective est de réduire le problème de la famine avec les grandes requêtes ; si après un tour, une requête n'est pas choisie alors dans le tour suivant sa possibilité d'être sélectionnée est plus grande.

La figure 3.8 a pour but d'illustrer cet algorithme :

A l'étape 1, il y a 4 requêtes A1, 3 requêtes A2 et une requête A3. Supposons que le temps d'exécution de chaque requête est 5 et le quantum original est 10. D'abord les requêtes sont agrégées dans la phase de pré-traitement. Puis un quantum de temps est proposé pour chaque requête. Puisque c'est la première fois que ces requêtes sont dans la queue, les quanta de temps proposés sont de 10. Les temps d'exécution des requêtes A1 et A2 étant de 20 et 15, elles ne sont pas sélectionnées. La requête A3 répondant à la condition du quanta est ordonnancée (son temps d'exécution est de 5).

A l'étape 2, quand la requête A3 est en train d'être exécutée, d'autres requêtes A2 et A3 arrivent. Les requêtes A2 sont agrégées dans la phase de pré-traitement. Puis un quantum de $10 * 2 = 20$ est proposé pour les requêtes A1 et A2 ($k = 2$) et un quantum de 10 est proposé pour la requête A3 (cette requête viens d'arriver, elle reçoit un quantum original). Les tailles des requêtes A1, A2, A3 sont 20, 25, 5 respectivement tandis que leurs quanta sont 20, 20, 10. Les requêtes A1 et A3 satisfont la condition d'exécution. l'ordre FIFO est employé, la requête composée A1 est traitée.

De manière similaire, quand les requêtes A1 sont en train d'être exécutées à l'étape 3, d'autres requêtes A2 et A3 continuent d'arriver. Après la phase de pré-traitement les tailles de requêtes composées A2 et A3 sont 30 et 20. La requête composée A2 se voit proposer un quantum de $20 * 2 = 40$ et la requête composé A3, un quantum de $10 * 2 = 20$. Les deux requêtes A2 et A3 satisfont la condition de sélection ; l'ordre FIFO est à nouveau appliqué, la requête composée A2 est sélectionnée selon l'ordre FIFO.

3.4.3 Contrainte de sous-requêtes

Dans cette section, nous attardons sur les contraintes liés aux systèmes de fichiers parallèles, les données sont souvent réparties sur plusieurs nœuds de stockages(*data striping*). Cette répartition nous permet de paralléliser des accès. Dans ce cas, données d'une requête peuvent être réparties sur plusieurs nœuds et nous devons considérer la relation entre ces sous requêtes. Par exemple, supposons que nous avons deux requêtes R1 (application A1) et R2 (application A2) de taille de 15 et 10 Ko et deux nœuds d'E/S ayant la taille de répartition (*"stripe size"*) de 5 Ko. Alors R1 est découpée en 3 sous requêtes et R2 est découpée en 2 sous requêtes (la taille de chaque sous-requête est 5 Ko). Supposons que nous utiliserons la stratégie WSJF sans vérifier les contraintes entre sous requêtes d'une même application (le temps d'exécution de chaque sous-requête correspond avec sa taille, soit 5 unités de temps). Alors, comme le montre la figure 3.9, le temps d'exécution de R1 est de 15 unités et de R2 est de 10 unités ; donc la totalité est de 25 unités. En tenant compte des contraintes de répartition, si les sous requêtes de A2 sur les deux nœuds d'E/S sont exécutés avant les sous requêtes de A1 (algorithme SOJF), le temps d'exécution de R2 est de 5 unités et de R1 de 15 ; dans ce cas la totalité est de 20 unités.

L'exemple ci-dessus montre l'importance de contraintes entre sous requêtes dans les systèmes de fichiers parallèles. Une idée simple est de synchroniser les sous requêtes d'une même application afin qu'elles soient exécutées au même moment. Une solution possible est d'appliquer un critère de sélection commun pour toutes les sous requêtes appartenant à une même application. Par exemple, nous pouvons appliquer la stratégie WSJF de manière indépendante sur chaque file d'E/S en prenant comme taille pour

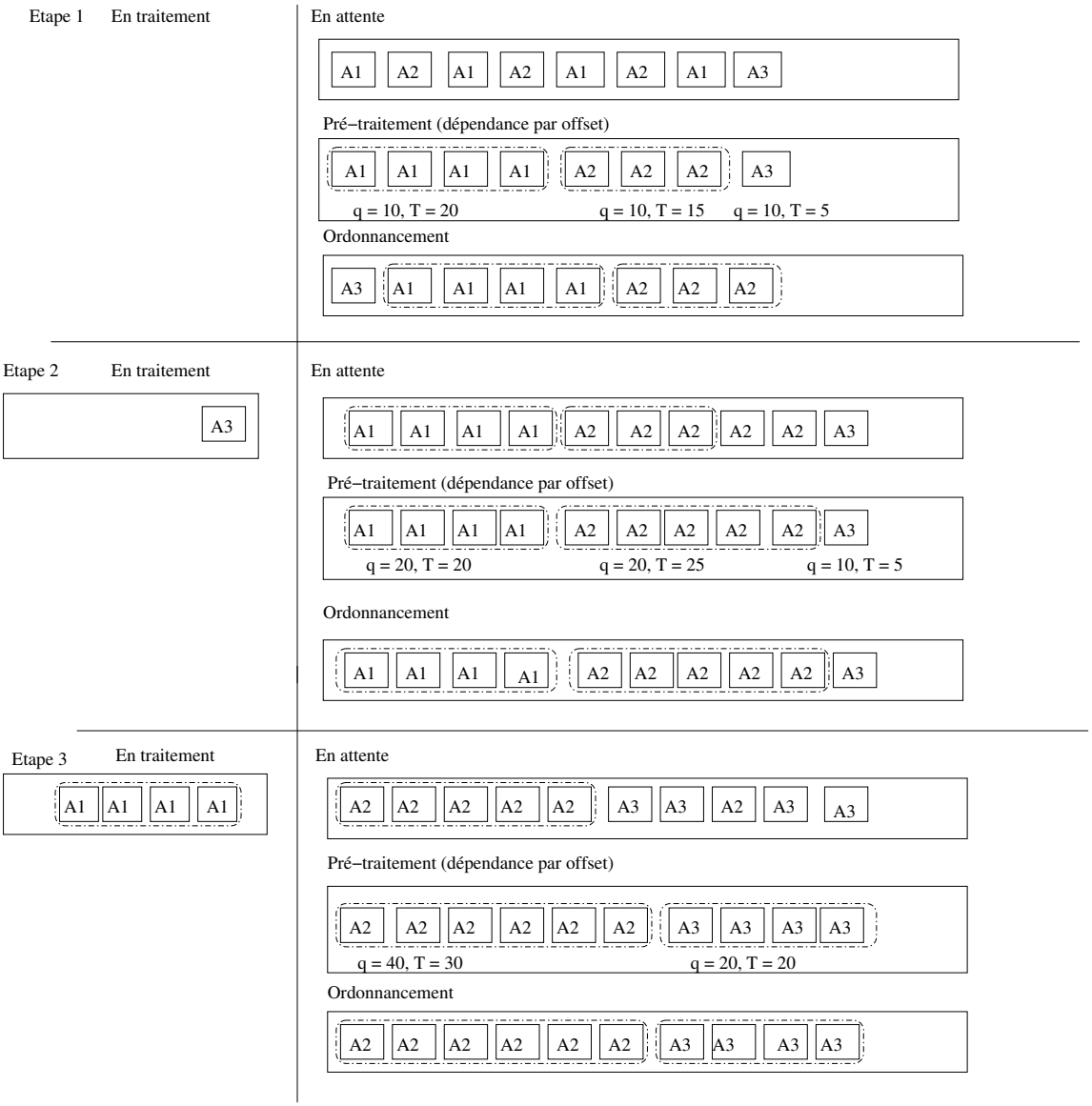


FIG. 3.8 – Une variante de MLF

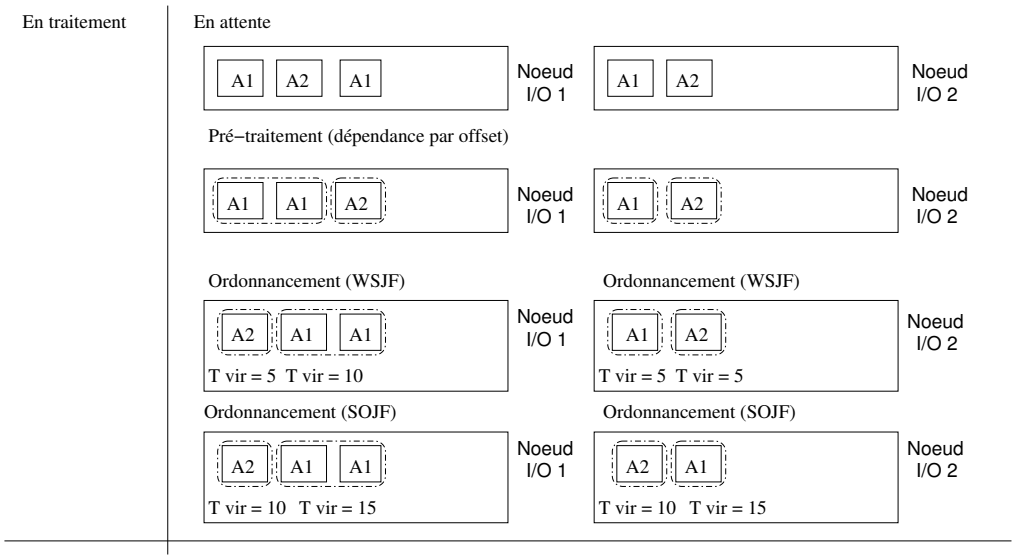


FIG. 3.9 – Contrainte de sous requêtes

le calcul du poids, la taille globale de la requête (au lieu de la taille des sous requêtes). Cette méthode est similaire à l'approche *"Shortest Outstanding I/O Demande Job First"* ou SOJF [44]. Dans l'exemple la requête R2 a la taille la plus petite parmi les deux requêtes ; l'approche SOJF sélectionne les sous requêtes de A2 (taille globale est 10) et puis les sous requêtes de A1 (taille globale est 15). Ce qui se révèle être une stratégie efficace. Cependant cette méthode, comme la méthode SJF, peut entraîner des problèmes de famine avec les requêtes ayant une grande taille. Nous proposons d'utiliser la méthode de pondération exploitée dans l'algorithme WSJF pour y remédier. A ce jour, nous n'avons pas intégrer ces derniers aspects à notre prototype, ce travail est une perspective supplémentaire du présent travail et nécessitera une étude plus approfondie.

Chapitre 4

Implantation et Evaluation

Le modèle présenté dans le chapitre précédent décrit une approche permettant la distribution en partie des concepts de la libraire aIOLi. Le prototype que nous allons présenter a été développé en langage C et nous a permis de mener nos évaluations. Il repose sur le modèle théorique décrit au chapitre précédent. Le programme permettant d'effectuer les décompositions afin d'évaluer ce nouveau prototype provient de précédent travaux, [34]. Il consiste en un programme MPI permettant la décomposition de fichier (simulation de multiplication parallèlisé de matrices)

4.1 Implantation du prototype

En s'appuyant sur l'approche employée lors du développement du premier prototype, nous avons développé deux modules : un module client permettant de surcharger les appels POSIX qui doit être *linké* à l'application client et un démon monolithique, déployé sur un noeud, afin de centraliser puis réguler les E/S. Comme nous l'avons abordé en 3.2.1, nous avons installé ce service sur le serveur NFS. Les requêtes clientes sont transmises par un canal `tcp` au serveur aIOLi. Chaque demande est stockée dans une queue correspondante selon le fichier qu'elle concerne, l'offset d'accès, sa taille et son heure d'arrivée. Comme nous le verrons par la suite, obtenir une prédiction juste de la durée de chaque requête n'a pas été évidente. Nous avons donc donner la possibilité à l'utilisateur d'activer ou non la prédiction : soit le traitement d'une nouvelle requête débute lors de la réception du message de fin de la précédente (cf. 3.3.2) soit elle débute sur la reception d'un signal délivré au serveur par une alarme temporisée selon la prédiction (cf. 3.3.3. A chaque fois que le serveur a à déterminer un nouvel ensemble de requêtes à traiter, il exécute la fonction correspondante à l'algorithme sélectionnée. Les deux stratégies d'ordonnancement abordées ont été implantées (cf. 3.4.2).

4.2 Expérimentations

Le système de test se compose de trois machines de la grappe "*idpot*" [1] (incluse dans *Grid5000*) : un serveur NFS (version 3, protocole TCP, taille de lecture 32Ko) et deux clients SMPs (1,5 Go de RAM, bi-processeurs IA32). Le débit disque du serveur est 54 Mo/sec (évalué par la commande `hdparm`).

Les tests corespondent aux cas que nous avons abordés dans 3.2 : coordination multi-nœuds dans laquelle une même application est déployée sur plusieurs nœuds SMPs et coordination multi-applicatifs dans laquelle plusieurs applications sont déployées sur plusieurs nœuds. Pour le premier cas, le programme de test est une application de décomposition d'un fichier distant de 2 Go stocké sur le serveur NFS. Cette application se compose de 8 processus déployé sur deux nœuds clients (4 processus sur chaque nœud). Le but de cette expérimentation est d'évaluer si les deux algorithmes permettent de découvrir les schémas d'accès parallèles. Dans la deuxième expérimentation, deux applications sont chacune déployées sur un nœud distinct. 4 processus par nœud). Chaque application décompose son propre fichier de 2 Go toujours

[1] `https://frontal38.imag.fr`

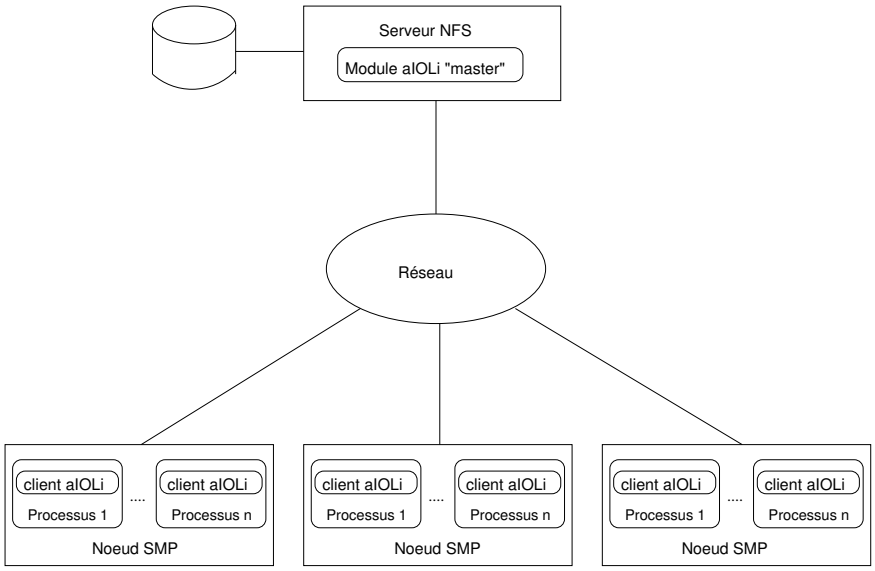


FIG. 4.1 – Architecture du système aIOLi

stockés sur le serveur NFS. Le but de cette expérimentation est d'observer le critère d'équitabilité entre les deux applications (différence de temps de complétion 4.4).

Une troisième expérimentation permettant de tester le critère d'intéractivité a été planifiée mais n'a pas encore pu être exécutée. Elle consiste à a lancer la récupération d'un fichier de 2 Go sur deux nœuds via la commande `cat`. Une troisième lecture d'un fichier de taille moindre (environ 100Mo) est réalisée pendant cette opération sur un troisième nœud. Le but étant d'analyser comment les temps d'accès au disques est réparti entre chaque application.

Afin d'éviter tout impact sur les performances (cache), chaque expérimentation utilise un fichier de 2Go différent. Toutes les décompositions sont exécutées avec différentes granularités de blocs de données afin d'analyser l'impact lié au nombre de messages transmis au serveur aIOLi.

4.3 Évaluation dans le cadre mono-applicatif

4.3.1 Détection des schémas d'accès parallèles

Lors des premiers lancements des tests, nous avons rapidement observé que l'utilisation des algorithmes WSJF ou MLF engendré des séries de requêtes ne pouvant point être agrégées. Le phénomène de décalage, présenté par la suite étant amplifié dans les deux approches.

Nous avons légèrement modifié les algorithmes afin d'essayer de favoriser les points de jonctions entre permettant des agrégations optimales. L'exemple suivant illustre ces propos : supposons que quatre requêtes de même taille doivent être envoyées au même serveur de données (`read(10,20), read(20,30), read(30,40), read(40,50)` sur le même fichier). [2] Supposons que les requêtes `read(10,20)` et `read(30,40)` arrivent sur le serveur aIOLi en même temps tandis que les requêtes `read(20,30)` et `read(40,50)` sont reçues un peu plus tard. Puisque les requêtes `read(10,20)` et `read(30,40)` ont la même taille, la même estampille de temps, et ne peuvent être agrégées (accès disjoints), les critères de sélection des algorithmes WSJF ou de MLF sont appliqués. Toutefois dans ce cas précis, les deux algorithmes ne peuvent déterminer la requête prioritaire puisqu'elles ont toutes deux les mêmes caractéristiques. Ainsi, si la requête `read(30,40)` est sélectionnée et exécuter avant, la queue contiendra à la prochaine étape, des requêtes disjointes : `read(10,20)` (précédemment en attente) et les nouvelles requêtes

[2] `read(x,y)` : lire de l'offset x à l'offset y dans le fichier.

`read(20,30)` et `read(40,50)` (qui viennent d'être insérées). Si au contraire la requête ayant offset plus petit, `read(10,20)`, est choisi, la queue contiendra les requêtes `read(20,30)`, `read(30,40)` et `read(40,50)` qui pourront être agréger en une requête `read(20,50)`. A partir de ces observations, une première idée est de choisir la requête ayant l'offset le plus petit parmi les requêtes disponibles tout en tenant compte du principal but de chaque algorithme. Dans l'algorithme WSJF, nous avons introduit un coefficient de jonction : parmi deux requêtes, une requête sera sélectionnée si son offset est inférieur à la deuxième requête et que sa taille virtuelle ne dépasse pas un ratio déterminé de celle de la deuxième (la valeur de 10% a été arbitrairement défini lors des évaluations). Dans l'algorithme MLF, nous avons modifié le critère de sélection FIFO employé lorsque deux requêtes ont le même degré de priorité en un critère basé sur l'offset (la plus petite valeur étant prioritaire).

4.3.2 Phénomène de décalage

Malgré les changements décrit au paragraphe précèdent, nous avons continué à observer un phénomène de type décalage. Grâce à une analyse plus fine, nous avons pu en déterminer la cause : l'ordre et la périodicité d'arrivée des requêtes en provenance des différents processus est dépendant d'un plus grand nombre de facteurs que dans la version précédente proposée par [34]. En effet, il avait été remarqué une forte dépendance entre les performances et l'ordonnancement des processus établi par le système Linux au sein d'un même nœud. Une fenêtre de réflexion similaire à une approche de type ordonnancement anticipé avait été intégré pour remédier en partie au problème. Dans notre cas, cette même dépendance est multiplié par le nombre de nœud participant à la décomposition puisque chaque stratégie d'ordonnancement pour chaque nœud est indépendante. De plus, les facteurs réseaux entre en jeux. Ainsi un processus peut délivrer une nouvelle requête dès que sa dernière a été traitée alors qu'un autre aura une periode de réemission beaucoup plus longue. La figure 4.2 illustre ce phénomène : quatre processus participent à la décomposition d'un même fichier. A l'étape 1, la requête de processus P0 arrive en retard par rapport aux autres requêtes (P1, P2, P3). Les requêtes de P1, P2, P3 sont contiguës et sont par conséquent agrégées au sein d'une même requête "virtuelle". Quand la requête P0 arrive au sein du serveur aIOLi, le premier message de PREPARATION pour la requête P1 a été déjà délivré, elle ne peut donc être agrégée au sein de cette requête "virtuelle" et est inséré dans la file d'attente. Elle est exécutée à l'étape 2 après le traitement de l'ensemble des requêtes composant la requête "virtuelle" . Les nouvelles requêtes de P1, P2 et P3 arrivent est sont agrégées dans étape 3. Le scénario se répète alors. Les accès en provenance de P0 ne peuvent jamais agréger avec les requêtes P1, P2, P3. Ce phénomène est d'autant plus amplifier si la périodicité d'envoi du processus P0 est longue.

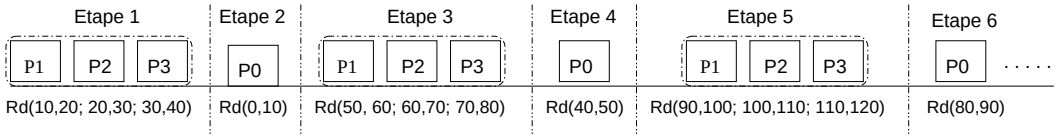


FIG. 4.2 – Requêtes décalées
L'ordre d'arrivée des messages à un impact sur le processus d'agrégation. Dans cet exemple, la requête PO ne va jamais être agrégée.

Cette exemple théorique est donnée à tire indicatif. D'un point de vue pratique, nous avons pu remarquer que dans les deux approches (WJSF ou MLF), ce décalage peut entraîner une diminution importante des performances : un processus peut nécessiter jusqu'à 20% de temps supplémentaire par rapport aux autres.

4.3.3 Problème de prédiction

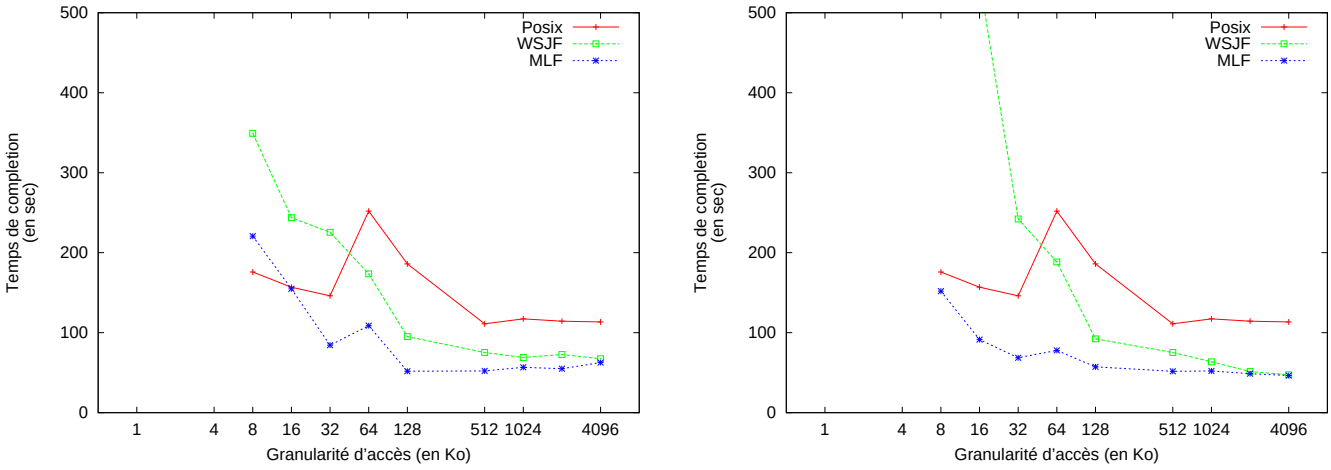
Le lancement de nouveaux tests nous a permis de détecter un problème supplémentaire lié à la méthode de prédiction : lorsqu'une prédiction s'avère erronée, elle va entraîner une dérive plus ou moins important de l'ensemble des prédictions suivantes. En effet, si la durée d'une requête est déterminée de manière fausse, le message PREPARATION de la requête suivante sera erronée et la nouvelle requête débutera trop

top. Une diminution non négligeable des performances pourra alors être perçue puisque les 2 accès seront traités en parallèle. De manière similaire, la seconde prédiction sera également fausse suite à la diminution des performances et va engendré une nouvelle dérive sur la suivante et ainsi de suite. La différence entre le temps de prédit et le temps réel va par conséquent grandir à chaque nouvelle erreur. Nous avons légèrement améliorer notre système afin de prendre en compte cette dérive à chaque nouvelle prédiction. Les résultats obtenues sont ceux présentés par la suite.

Néanmoins, pour les petites tailles, nous n'avons pas réussi à avoir une prédiction proche des temps réels. Ce problème a été abordé dans [45]. Des discussions ont commencé avec différentes personnes de la thématique de l'évaluation et des prédictions de performances du laboratoire ID afin d'enrichir notre modèle et d'essayer de fournir une approche utilisable quelque soit la granularité d'accès.

4.3.4 Résultats

Dans cette première phase, nous avons testé le nouveau prototype afin d'en évaluer son comportement dans le cadre d'une coordination multi-nœuds d'une même application. Les résultats apparaîssent au sein de la figure 4.3 (a) pour le mode sans prédiction et 4.3 (b) pour le mode avec prédiction.



(a) Sans prédiction

(b) Avec prédiction

FIG. 4.3 – Décomposition d'un fichier de 2Go incluant 8 instances MPI
Au contraire de (b), la figure (a) représente l'expérimentation sans utiliser la méthode de prédiction de données. Chaque nouvelle requête est exécutée lorsque la précédente se termine.

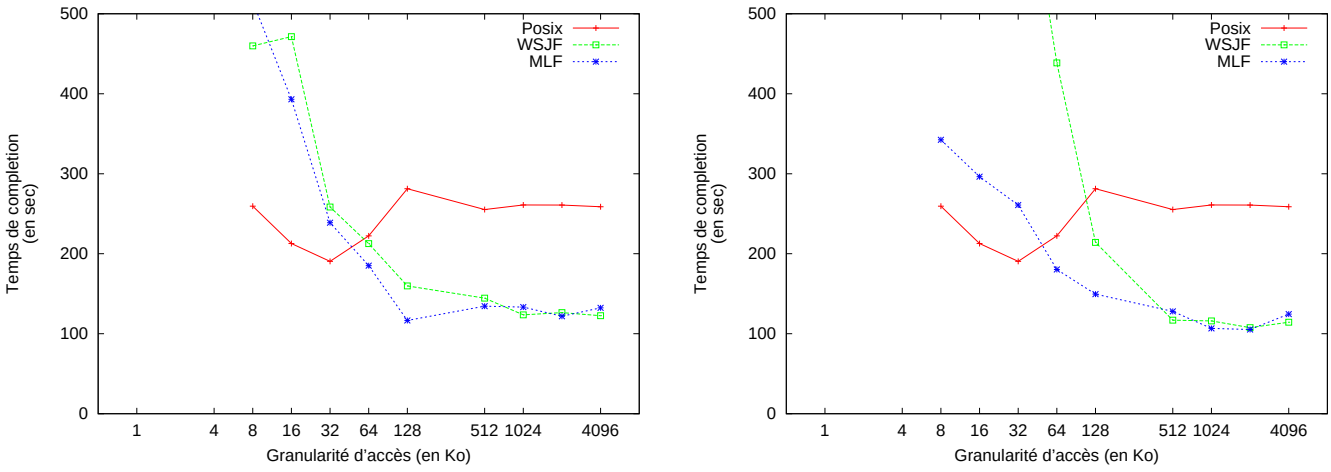
Pour la méthode sans prédiction, aucun apport est fournit par les deux algorithmes pour les accès inférieures à 32Ko. Au contraire, un surcoût non négligeable est présent en comparaison à l'approche POSIX standard (basé sur NFS). Plusieurs points peuvent expliquer ces pertes : Tout d'abord la granularité d'accès du client NFS est de 32Ko, ainsi chaque demande comprise entre 1Ko et 32Ko engendre une requête NFS read de 32Ko. De ce fait, nombreuses requêtes sont directement satisfaites par le cache client. Dans les approches aIOLi (WSJF et MLF), toute demande et ce quelque soit la granularité, est transmise au serveur aIOLi ou elle est régulée. Ainsi la première requête se finit au minimum après le temps nécessaire à l'échange des 3 messages : DEMANDE (du client au serveur) / PREPARATION (du serveur au client) / FINI (du client au serveur) et les suivantes requiert le temps de 2 messages : PREPARATION / FINI (le message DEMANDE étant récouvert par les requêtes précédentes). Soit sur l'architecture idpot, un délai de synchronisation moyen de 2 fois la latence (environ 100μs) et ce même si la donnée est présente dans le cache. Ainsi, pour une décomposition de 2Go à 8Ko, il faut un peu plus de 256000 requêtes. Par conséquent, environ 27 secondes sont déjà nécessaire pour réaliser la synchronisation. L'approche MLF profite assez bien des effets caches puisqu'elle ajoute une faible surcoût par rapport à l'approche standard (en moyenne 220 secondes pour 8Ko et 180 sec pour l'approche standard). Par contre, nous pouvons observer que le

coefficient de jonction utilisé dans WSJF n'est pas initialisé à une valeur suffisante. Les requêtes étant mal réordonnées (mauvaise détéction des schémas d'accès parallèles), les clients ne bénéficient pas du cache et au contraire génèrent une multitude d'accès disjoints. Tout au long de l'expérience, l'algorithme MLF se comporte mieux car il choisit toujours la requête de plus petit offset.
A partir de 32Ko, l'approche aIOLi devient performante pour les deux algorithmes ; les requêtes Posix ne pouvant plus être directement satisfaites par le cache client, elles entrent en conflit sur le serveur de fichiers qui doit les gérer en parallèle. Plus la granularité est important plus notre système est performant ; le nombre de requêtes nécessaire à la décomposition devenant moindre (pour 4Mo seulement 512), les les délais de synchronisations deviennent transparent.

L'intérêt d'une approche avec prédiction de temps est de rendre transparent la synchronisation de deux requêtes sur le serveur de stockage (cf. 3.3) Nous voyons que c'est en partie résolue pour l'approche MLF performante dès 8Ko. Dans le cas d'utilisation de la méthode de prédiction, l'algorithme MLF est meilleur que l'approche traditionnelle POSIX quelque soit la granularité (de 8Ko à 4 Mo). Cependant, le choix non efficace de requête dans la stratégie WSJF ne permet pas de bénéficier de la prédiction et globalement l'algorithme apporte les mêmes résultats qu'en 4.3 (a).

4.4 Évaluation dans le cadre multi-applicatif

Le but de cette expérimentation a été d'évaluer notre approche dans le cadre de l'exécution de plusieurs applications parallèles sur une grappe. L'application utilisée lors du précédent test est exécuté sur 2 nœuds distincts (4*2 instances MPI). Chaque application réalise une décomposition sur un fichier de 2Go stocké tous deux sur un serveur NFS.
Les résultats sont présentés dans la figure 4.4.



(a) Sans prédiction

(b) Avec prédiction

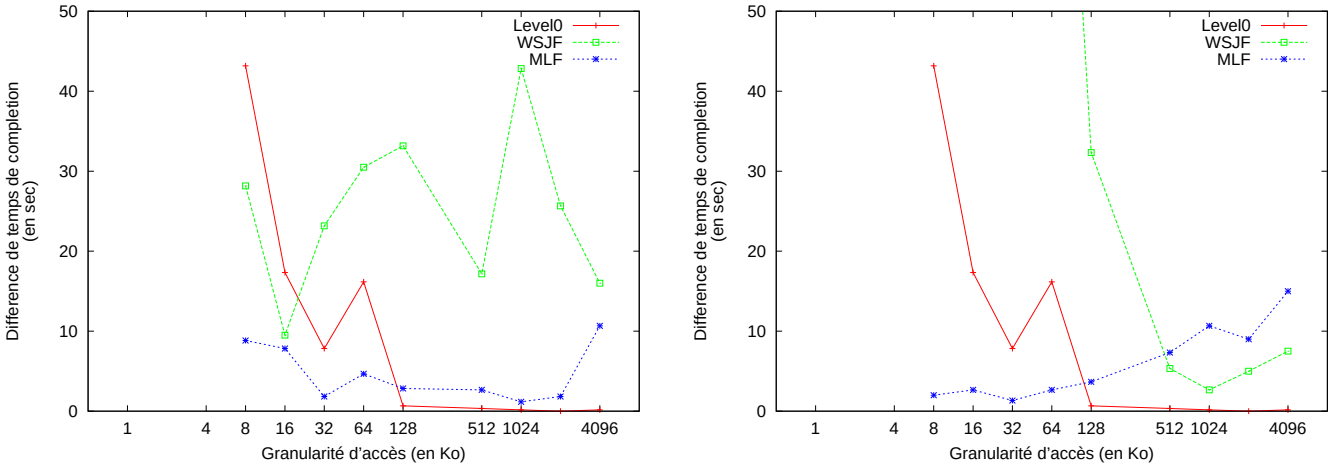
FIG. 4.4 – Décomposition de 2 fichiers de 2Go incluant 4*2 instances MPI

Comme précédemment, l'approche sans prédiction engendre des coûts trop important en délai de synchronisation et n'est donc pas favorable pour les accès de faible granularité. Il est cependant intéressant de noter que cette fois ci, l'algorithme MLF semble entraîner le plus de surcoût. Ce qui en d'autres termes, signifie qu'en cadre multi-applicatif, et pour les petits accès, l'approche MLF "bascule" plus souvent entre les deux applications diminuant ainsi l'efficacité du cache NFS client. Cette observation sera confirmé par la suite lorsque nous analyserons le critère d'équité entre les applications.

Dans l'approche avec prédiction, nous pouvons remarqué qu'un gain pour MLF est à nouveau apporté pour les accès de petites tailles. Néanmoins pour les accès plus conséquents, nous pouvons observer une

légère baisse des performances remettant en cause l'approche avec prédiction. Il serait intéressant d'approfondir ce comportement et analyser si la dégradation des prédictions est plus forte en mode multi-applicatif (faute de temps, nous n'avons pu abordé cet aspect). Il en est notamment de même pour l'approche WSJF.

Un des buts de cette expérimentation est de comparer, certes, l'apport réalisé par les stratégies d'ordonnancement WSJF et MLF mais également d'étudier le critère d'équité entre les deux applications. Lors du choix du critère d'optimisation dans l'architecture aIOLi (cf. 3.4.1), nous avions mentionné l'importance de trouver un compromis acceptable entre le critère efficacité (performance, maximisation des débits) et le critère interactivité et équité. La figure 4.5 s'attarde sur ce dernier critère qu'est l'équité. L'écart entre les deux temps de complétion de chacune des applications est mesuré.



(a) Sans prédiction

(b) Avec prédiction

FIG. 4.5 – Décomposition de 2 fichiers de 2Go incluant 4*2 instances MPI
Critère d'équité, l'écart entre les temps de completion restent largement acceptable pour l'algorithme MLF par rapport au gain apporté (cf. figure4.4).

D'un point de vue global, l'approche standard POSIX est peu équitable pour les petits accès puisqu'ils sont traitées de manière totalement indépendantes et sont réparties sur les 2 * 2Go de données (certains requêtes sont favorisées par rapport à d'autres). L'approche WSJF semble subir les effets du décalage (cf. 4.3.2) et donne des résultats qu'il est nécessaire d'approfondir. Finalement, l'algorithme MLF fournit un très bon rapport équité / performance. En effet, seules 10 secondes séparent les deux applications à 4 Mo pour un gain proche de 50%.

Chapitre 5

Bilan

Dans cette partie, nous avons présenté une distribution des concepts proposés par la librairie d'E/S "aIOLi". Les principes, les contraintes et les solutions choisies pour la mise en œuvre d'une coordination multi-nœuds mais aussi les limites d'une telle approche ont été abordés. Les résultats expérimentaux obtenus ont montré les gains pouvant être apportés par une approche basée sur une régulation et un ordonnancement adéquat des requêtes d'E/S . Les principales difficultés se sont situées autour de la synchronisation des requêtes sur le serveur de fichiers et la proposition d'algorithmes d'ordonnancements bi-critères (performance et équité).

L'utilisation de techniques à base de prédiction qui devrait, d'un point de vue théorique, permettre de rendre transparent le délai de synchronisation, s'est avérée nettement plus complexe en pratique. La difficulté d'une prédiction rigoureuse pour les accès de faible tailles en est la principale limitation. De plus, chaque prédiction erronée, entraine une dégradation redondante et progressive sur l'ensemble des prédictions suivantes. Néanmoins, il nous semble intéressant d'approfondir cette étude en analysant un modèle exploitant des techniques similaires à celles employées dans l'outil *Network Weather Services*.

La découverte des schémas d'accès fortement exploité au sein de la première version du système aIOLi s'est également avérée plus complexes. Les nombreux facteurs comme les ordonnancement systèmes de chaque nœud (*linux scheduler*), la latence et la congestion des réseaux ont un impact considérable sur l'arrivée des demandes au sein du service aIOLi. Des améliorations prenant en compte une partie de l'historique des requêtes traitées pour chaque fichier pourraient permettre une meilleure découverte et donc de meilleures performances.
L'utilisation d'un simple facteur de jonction au sein de l'algorithme WSJF n'a pas été par exemple suffisant.
D'un point de vue global, il paraît aujourd'hui judicieux d'exploiter une ordonnancement spécifique au sein d'un même fichier et des ordonnancements de type WSJF ou MLF entre les fichiers. L'approche MLF a déjà montré que pour des accès de tailles conséquentes, le ratio équité/gain était très prometteur. L'algorithme WSJF n'a pas été assez analysé et l'ensemble des résultats fournit lors de l'étude multi-applicatif a malheureusement était faussé par un mauvais choix en intra-fichiers. La mise en place d'un troisième test à base de lecture séquentielle via l'utilisation de la commande `cat` devrait fournir des résultats plus "parlant".
Enfin, il sera nécessaire d'inclure au sein des algorithmes d'ordonnancements, les contraintes liées au serveur de fichiers parallèles et éviter ainsi tout choix erroné. Le système ainsi formé devrait être applicable à un plus large panel de systèmes de fichiers.

Parallèlement, il sera intéressant d'évaluer ce même prototype mais en déployant notre démon aIOLi sur un nœud différent du serveur de fichiers. En effet, nous n'avons pas pu étudier l'influence du démon aIOLi sur l'exécution du serveur NFS (et réciproquement). Il serait intéressant de connaître cet impact et de le comparer à une approche de type hiérarchique, plus prometteuse pour un passage à l'échelle efficace. Une étude d'une architecture s'appuyant sur la mise en place d'un démon aIOLi de niveau 1 par sous réseau (par switch) puis de niveau 2 par grappe (par routeur) fournirait des observations intéressantes et fera partie de nos objectifs par la suite.

Chapitre 6

Conclusion

Le problème des Entrées/Sorties est un thème de recherche important dans les systèmes informatiques. L'apparition des nouveaux systèmes parallèles comme les grappes ou les grilles l'ont rendu prépondérant. L'exécution d'applications scientifiques "*HPC I/O intensive*" nécessitent des outils fiables efficaces et simples d'utilisations. Plusieurs solutions (méthodes théoriques, systèmes de fichiers "parallèles", librairies spécialisées) ont été présentés, y compris la solution aIOLi - une librairie d'Entrées/Sorties parallèles pour nœud SMP au sein d'une grappe.
Le principal intérêt de l'étude menée, a été d'analyser et de proposer une solution permettant de distribuer à l'échelle d'une grappe cette librairie. Par ailleurs, nous avons ajouté aux optimisations déjà présentes, des algorithmes d'ordonnancement permettant de prendre en compte la concurrence entre les accès provenant d'applications distincts.
Enfin, la mise en place d'une agrégation physique des requêtes au sein d'un nœud maître contribuerait à améliorer nettement les performances en plus de diminuer le nombre de messages réseaux nécessaire pour la synchronisation des accès. Toutefois, cette approche implique de nombreuses contraintes (gestion de la cohérence des données, réplication, système de "*callbacks*" pour invalider les caches, ...) à la différence du mécanisme exploité au sein de la version précédente d'aIOLi. Nous avons donc choisi de laisser dans un premier temps ce point de côté mais il serait intéressant de l'approfondir également

L'utilisation croissante des grappes par les scientifiques mais aussi et de plus en plus, par les industriels a déjà entraîné de nouvelles contraintes dans les gros centres de calcul comme l'IDRIS[1] ou le CEA. La mise à disposition d'un unique système de fichiers à la NFS ne répond plus aux besoins que ce soit en terme de performances (débit), de sécurité (protection des données inter-utilisateurs) ou encore en terme de tolérance aux pannes et pérennité de l'information. La donnée n'est plus "figée" sur un unique disque ; elle transite depuis les serveurs de sauvegarde aux serveurs de mises à disposition pour enfin atteindre les serveurs d'exploitation présents dans ces nouvelles architectures hautement distribuées. La quantité de données ainsi transférer est considérable. Il est donc primordiale, qu'à chacun des niveaux, le système réponde à des critères bien spécifiques :

- gestion fine et efficace des accès parallèles et/ou concurrents pour les serveurs d'exploitations, sémantique forte ;
- parallélisation des accès et agrégation des débits entre les serveurs d'exploitation et de mises à disposition (à la GrdiFTP), sémantique relâchée ;
- réplication des données sur les serveurs de sauvegarde , sémantique faible
- ....

L'engouement de plus en plus prononcé pour les grilles (projet GRID5000) a lui aussi commencé à imposer de nouvelles contraintes en termes de passage à l'échelle. De nombreux travaux scientifiques sont actuellement réalisés dans ce domaine. Ils tentent de répondre aux problèmes et contraintes tels que l'ordonnancement, les interfaces de programmation ainsi que l'accès et le partage à distance de l'ensemble des ressources. L'étude et l'analyse d'approche tentant d'apporter des optimisations à chaque niveau de la

[1] Centre de calcul du CNRS.

hiérarchie nous paraît intéressante et sera notre principal objectif par la suite. Le but étant d'exploiter tous les points de concentration physique (switch, routeur, …) pour appliquer des optimisations.

Bibliographie

[1] G.M. Amdahl. Validity of the single-processor approach to achieving large scale computing capabilities. pages 483–485, 1967.

[2] David A. Patterson et John L. Hennessy. *Computer Architecture : A Quantitative Approach*. Kaufmann, 1996. HEN j2 96 :1 1.Ex.

[3] I. Foster et C. Kesselman. *The grid : blueprint for a new computing infrastructure*. Morgan Kaufmann, 1999.

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

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

[6] John M. May. *Parallel I/O for High Performance Computing*. Morgan Kaufmann, 2001.

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

[8] Bill Nitzberg et Virginia Lo. Collective buffering : Improving parallel I/O performance. In Hai Jin, Toni Cortes et Rajkumar Buyya, editors, *High Performance Mass Storage and Parallel I/O : Technologies and Applications*, chapter 19, pages 271–281. IEEE Computer Society Press and Wiley, New York, NY, 2001.

[9] Adrien Lebre. Haute performance et entrées/sorties au sein des applications parallèles. Mars 2004.

[10] S. Iyer et P. Druschel. Anticipatory scheduling : A disk scheduling framework to overcome deceptive idleness in synchronous i/o. *Appear in the 18th ACM Symposium on Operating Systems Principles*, 2001.

[11] Avery Ching, Alok Choudhary, Kenin Coloma, Wei keng Liao, Robert Ross et William Gropp. Noncontiguous I/O accesses through MPI-IO. In *Proceedings of the Third IEEE/ACM International Symposium on Cluster Computing and the Grid*, pages 104–111, Tokyo, Japan, May 2003. IEEE Computer Society Press.

[12] Rajeev Thakur, William Gropp et 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.

[13] 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.

[14] K. Pruhs, J. Sgall et E. Torng. Online scheduling. In *Hanbook of Scheduling*, chapter 15. CRC Press, 2004.

[15] Nikhil Bansal. *Algorithms for Flow Time Scheduling*. PhD thesis, Dept. of Computer Science, Carnegie Mellon University, 2003.

[16] B. Kalyanasundaram et K. Pruhs. Speed is as powerful as clairvoyant. *Journal of the ACM, Vol. 47, No. 4*, pages 617–643, July 2000.

[17] B. Kalyanasundaram et K. Pruhs. Minimizing flow time nonclairvoyantly. In *Proceedings of the 38th Annual Symposium on Foundations of Computer Science (FOCS '97)*, 1997.

[18] C. Chekuri, A. Goel, S. Khanna et A. Kumar. Multi-processor scheduling to minimize flow time with epsilon resource augmentation. In *in Proc. of the 36th annual ACM Symposium on Theory of Computing (STOC 2004)*, 2004.

[19] A. S. Tanenbaum. *Modern operating systems*. Prentice Hall, 2001.

[20] Robert B. Ross. *Reactive Scheduling For Parallel I/O Systems*. PhD thesis, Clemson University, 2000.

[21] Philip H. Carns, Walter B. Ligon III, Robert B. Ross et 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.

[22] Heinz Stockinger. Dictionary on parallel input/output. Master's thesis, Department of Data Engineering, University of Vienna, February 1998.

[23] Rainer Hubovsky et Florian Kunz. Dealing with massive data : from parallel i/o to grid i/o. Master's thesis, Vienna University of Technology, Vienna, Austria, January 2004.

[24] Florin Isaila. *An overview of file system architectures.*, chapter 13, pages 273–289. Lecture Notes in Computer Science. March 2003.

[25] Network file system (nfs) version 4 protocol, 2003.

[26] Dean Hildebrand et Peter Honeyman. Exporting storage systems in a scalable manner with pnfs. In *Proceedings of the 22nd IEEE/13th NASA Goddard Conference on Mass Storage Systems and Technologies (MSST)*, 2005.

[27] Florin Isaila et Walter F. Tichy. Clusterfile : a flexible physical layout parallel file system. Jul 2003.

[28] Florin Isaila, Guido Malpohl, Vlad Olaru, Gabor Szeder et Walter Tichy. Integrating collective I/O and cooperative caching into the "clusterfile" parallel file system. In *Proceedings of the 18th Annual International Conference on Supercomputing*, pages 58–67, Sain-Malo, France, July 2004. ACM Press.

[29] MPI I/O (website).

[30] Kent E. Seamons. *Panda : Fast Access to Persistent Arrays Using High Level Interfaces and Server Directed Input/Output*. PhD thesis, University of Illinois at Urbana-Champaign, May 1996.

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

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

[33] Bill Allcock, Joe Bester, John Bresnahan, Ann L. Chervenak, Ian Foster, Carl Kesselman, Sam Meder, Veronika Nefedova, Darcy Quesnel et Steven Tuecke. Data management and transfer in high-performance computational grid environments. *Parallel Computing*, 28(5) :749–771, May 2002.

[34] A. Lebre et Y. Denneulin. aioli : An input/output library for cluster of smp. Decembre 2004.

[35] A. Lebre et Y. Denneulin. aioli : gestion des entrées/sorties parallèles dans les grappes smp. *Rapport de Recherche*, Mars 2005.

[36] M. Singhal et N. G. Shivaratri. *Advanced concepts in operating systems*. McGraw-Hill, 1994.

[37] I. Suzuki et T. Kazami. A distributed mutual exclusion algorithm. *ACM Trans. on Computer Systems (TOCS) 3*, Nov. 1985.

[38] M. Raynal. A tree-based algorithm for distributed mutual exclusion. *ACM Trans. on Computer Systems (TOCS)*, 1989.

[39] L. Lamport. Time, clocks and ordering of events in distributed systems. *Communications of the ACM*, July 1978.

[40] G. Ricart et A. K. Agrawala. An optimal algorithm for mutual exclusion in computer networks. *Communications of the ACM*, January 1981.

[41] M. Raynal. A simple taxonomy for distributed mutual exclusion algorithms. *Rapport de Recherche*, Novembre 1990.

[42] R. Wolski. Dynamically forecasting network performance using the network weather service. *appeared in Cluster Computing : Networks, Software Tools, and Applications*, Jan. 1998.

[43] M. Seltzer, P. Chen et J. Ousterhout. Disk scheduling revisited. In *Proceedings of USENIX*, pages 313–323, 1990.

[44] F. Chen et S. Majumdar. Performance of parallel i/o scheduling strategies on a network of workstations. *Eighth International Conference on Parallel and Distributed Systems*, 2001.

[45] Manish Sharma et John W. Byers. How well does file size predict wide-area transfer time ? In *Proceedings of the 2002 Globecom Global Internet Symposium*, Taipei, Taiwan, October 2002.