aIOLi : Contrôle, Ordonnancement et Régulation des Accès aux Données Persistantes dans les Environnements Multi-applicatifs Haute Performance

Adrien Lebre

INSTITUT NATIONAL POLYTECHNIQUE DE GRENOBLE

N° attribué par la bibliothèque
/_/_/_/_/_/_/_/_/_/

THÈSE

pour obtenir le grade de

Docteur de L'Institut National Polytechnique de Grenoble (INPG)

Spécialité : «Informatique : Systèmes et Logiciels»

préparée au laboratoire Informatique et Distribution
dans le cadre de ***l'École Doctorale «Mathématiques, Sciences et Technologies de l'Information, Informatique»***

présentée et soutenue publiquement

par

Adrien Lebre

le 15 septembre 2006

---

alOLi :
Contrôle, Ordonnancement et Régulation des Accès aux Données Persistantes dans les Environnements Multi-applicatifs Haute Performance

---

Directeur de thèse : Brigitte PLATEAU

JURY

JACQUES MOSSIÈRE	INPG	Président
CHRISTINE MORIN	IRISA	Rapporteur
PIERRE SENS	U. Paris VI	Rapporteur
PASCALE ROSSÉ-LAURENT	BULL SA	Examinateur
YVES DENNEULIN	INPG	Co-Directeur de thèse
BRIGITTE PLATEAU	INPG	Directeur de thèse

La mise en page a été réalisée avec LaTeX. La police utilisée pour le corps du texte est une police avec sérif appelée Palatino. Les schémas ont été réalisés avec `Xfig` *et les courbes avec* `Gnuplot`*. Le système utilisé pour la rédaction a été constitué d'installations de Debian GNU/Linux.*

Version : 2006-12-05 17:50

Chacun sa route, chacun son chemin
Chacun son rève, chacun son destin !
Tonton David.

REMERCIEMENTS

Ça y est c'est ***Le Moment !*** Celui qui marque *Le Point Final* de ces longues années d'études. Les remerciements, par habitude, sont rédigés après la présentation des travaux. Je ne fais pas défaut à la règle : *ça y est je suis docteur ;) ! ! !*

Durant ces longues années, j'ai rencontré de nombreuses personnes qui ont toutes, à leur manière, contribué à l'aboutissement de ces travaux. A défaut d'être atypique, j'ai choisi de les remercier en remontant les années depuis cette date du 15 septembre 2006.

Tout d'abord, je tiens à remercier mon jury :

- **Jacques Mossière**, qui m'a fait l'honneur de présider ce jury et l'amabilité de chambouler son emploi du temps dans les derniers instants afin de permettre que cette soutenance se passe dans les meilleures conditions ;
- **Pierre Sens** et **Christine Morin** qui ont accepté de prendre du temps sur leur période estivale pour relire mon manuscrit : «Les modifications apportées suite à vos remarques améliorent la qualité du document en facilitant sa lecture et sa compréhension, je vous en remercie !» ;
- **Brigitte Plateau** qui a également modifié son emploi du temps pour être présente à cette soutenance. Plus personnellement, je souhaiterais également la remercier d'avoir accepté le rôle de directeur de thèse pendant ces 3 années, «je n'aurais tout simplement pas pu réaliser cette aventure sans ton rôle, merci !»

Cette thèse s'est déroulée principalement autour d'une convention CIFRE entre la division Open Source de BULL située à Echirolles et le laboratoire ID-IMAG situé en région Grenobloise. Je tiens à remercier globalement ces deux instituts et leurs tutelles associées pour m'avoir permis de réaliser cette thèse dans de si exceptionnelles conditions. Je m'aperçois, encore plus *a posteriori*, de la chance presque insolente que j'ai eu.
D'une manière plus particulière, je tiens à remercier les deux personnes qui ont encadré ces travaux :

- **Pascale Rossé-Laurent**, co-encadrant BULL, pour une quantité de chose et peut être la plus importante : m'avoir rappelé à maintes reprises qu'une idée non structurée reste simplement une idée, «Merci Pascale» ;
- **Yves Denneulin**, co-encadrant ID-IMAG, pour la confiance qu'il m'a accordée durant ces années, la liberté d'action et ses interventions souvent nécessaires pour canaliser mon enthousiasme, «Merci à toi, j'espère que nous n'avons pas fini de collaborer !»

Au delà des personnes qui ont tenu des positions «officielles» autour de cette thèse, je souhaiterais remercier l'ensemble des personnes qui ont contribué à la réalisation du projet *aIOLi* ainsi qu'à la rédaction de ce manuscrit :

- A ceux que par habitude je nommais «les théoriciens», ils ont apporté la colorisation *ordonnancement* au projet :
  - **Lionel** pour m'avoir indiqué que mon problème était réellement complexe et m'avoir aidé à réaliser une première ébauche du modèle ;

  - **Guillaume**, pour m'avoir indiqué que j'étais vraiment complexe ;) et avoir accepté de collaborer durant ces deux dernières années autour du projet.
  - **Arnaud**, dernier entrant, pour m'avoir expliqué que l'équité c'était sacrément complexe ! Merci d'avoir pris le temps de m'aider à structurer la partie ordo dans ce manuscrit. Je t'en suis largement redevable.

  A tous trois, je vous dit «Merci !»
- Aux stagiaires, **Trung**, **Przemyslaw**, **Fabien**, «Merci». De manière plus particulière, je voudrais adresser un merci à Przemek, étudiant polonais : «Dziekuje przyjaciele»
- A **Pierre Lombard**, parce qu'il le vaut bien ;) «Merci !»
- A l'ensemble des chercheurs que je n'ai pas encore remercié et qui de manière directe ou indirecte ont permis que le projet *aIOLi* mûrisse : **Jean-Marc**, **Bruno G.**, **Denis T.**, **Titou**, **Vania**, **Olivier R.**, **Jean-Francois**, **Vincent D.** ou encore **Greg** «Merci également à vous !» Je tiens à associer à ces remerciements **Jacques Briat** qui m'a également beaucoup aidé dans la première étape de cette thèse, «Merci et bon vent !»
- Aux personnes qui ont eu la gentillesse (et pour certains la patience) de relire une ou plusieurs parties de ce manuscrit : **Lucas**, **Pierre**, **Guillaume**, **Fabien**, **Cécile**, **Laurent**, **Vania**, **Arnaud**, **Olivier**, **mon frère aîné** et tous ceux que j'oublie, à vous tous un grand et sincère merci.

Ces 6 années passées au laboratoire ont été très riche humainement, je garderais d'intarissables souvenirs :

- d'un bureau, le 118, de **Julien** et nos délires, de **Lucas** et nos fous rires, de **Jaroslaw** et de son polonais, de chacune des personnes qui a fait que pendant ces années, ce bureau a toujours été un endroit de plénitude ;)
- d'une machine à café trop souvent en panne ces dernières années mais primordiale pour permettre de faire fructifier une quantité de théorie plus ou moins vérifiée mais tellement sympathique. Merci aux compagnons des pauses thé et/ou café : les matinales **Jean Marc**, **Denis N.**, **Denis T.**, les après-repas **Bruno G.**, **Bruno R.**, **Jean Louis**, **Joelle**, **Yves**, **Lucas**, **Feryal**, **Olivier**, **Guillaume**, ...
- de thésards, d'ingénieurs ou encore de stagiaires qui font que le laboratoire est un environnement agréable, convivial, multi-ethnique et propice à la recherche. Merci **Euloge**, **Said Maurico**, **Rafaël**, **Laurent**, **Samir**, **Ihab**, **Krzysztof**, **Wilfrid**, **Nico**, **Olivier V.**, **Brice**, **Leonardo** ...merci à vous tous !
- des assistantes qui ont du dans de nombreux cas jongler avec mes difficultés à comprendre la signification du mot «délai». Merci à **Annie-Claude**, **Barta**, **Marion**, **Anne-Laure**, **Evelyne**, ...

L'étape préalable à la thèse est le D.E.A, je tiens à te remercier également **Philippe Augerat** qui en plus de m'avoir fait découvrir les problématiques liées aux grappes, m'a apporté son soutien dans la mise en place d'un financement CIFRE «Merci à toi !»

Tout au long de mon cursus, j'ai rencontré plusieurs personnes qui m'ont fait découvrir l'informatique et qui ont fait qu'à mon tour j'en suis passionné. Je tiens à remercier tout particulièrement les enseignants du BTS Info Indus du lycée ASTIER sur la commune d'AUBENAS *«ça y est j'ai fini, merci **Toufike**, **Fabrice**, **Laurent**, **Philippe** ! j'espère que rien n'a changé et que la pédagogie employée est restée identique ! merci à vous.»*

Mes derniers remerciements vont à mes amis proches (**Teisso**, **Guillaume**, **la bande RIUP**, **Adil et tout les autres d'Astier**) ainsi qu'à **ma famille** pour m'avoir supporté dans tous les sens du termes. Merci à vous pour tant de choses.

Il me semble impossible de ne pas terminer ces remerciements sans te citer. Je pense que je ne trouverai jamais les mots adéquats pour exprimer tout le respect et la reconnaissance que j'ai pour toi. Merci **Do**,
Pour moi, pour toi, pour elle et pour ceux à venir !

Ça y est c'est *Le point Final* pour moi et *Le Commencement* pour vous, bonne lecture !

TABLE DES MATIÈRES

TABLE DES FIGURES

INTRODUCTION

Contexte général

L'informatique a fourni le moyen de traiter rapidement et de conserver des informations dans de nombreux domaines d'activités : industriel, économique ou social. L'évolution permanente des technologies qui la compose modifie sans cesse les manières dont nous l'exploitons. Par exemple, les nombreux travaux de recherche menés depuis une dizaine d'années autour des grappes [1] [SSB+95] puis des grilles (grappes de grappes) [FK99] ont eu un impact considérable sur les nouveaux développements parallèles.

Plusieurs avancées significatives ont été réalisées dans ce domaine aussi bien en terme d'ordonnancement, d'algorithmique parallèle, d'environnement de programmation ou encore de simplicité d'utilisation. Afin de tirer profit de ces améliorations, un grand nombre d'applications scientifiques (climatologie, biologie moléculaire, génomique, physique des hautes énergies ou encore traitement d'images ...) a été développé ou redéveloppé. Bénéficiant des grandes puissances de calcul disponibles, ces nouveaux systèmes de résolution s'appuient de plus en plus sur une grande quantité de données et utilisent de manière intensive les systèmes de stockage sous-jacents. De ce fait, leurs performances sont intimement liées aux méthodes et aux politiques de gestion de données choisies. C'est sur ces aspects que se concentre ce mémoire.

En 1967, Gene Amdahl [Amd67] montrait déjà que le gain maximal pouvant être obtenu lors de la parallélisation d'un programme est limité par les performances du sous-système le plus «lent». Ainsi, une application dont le temps est réparti de la manière suivante : 90% calcul et 10% d'entrées/sorties [2] , sera toujours, quelques soit le nombre de processeurs utilisé, dépendante de la phase des E/S. Partant de ce constat, Amdahl a suggéré un principe de base : une machine est équilibrée lorsqu'elle possède 1 Mo de mémoire vive et une bande passante pour les E/S de 1 Mbits/s pour une puissance *CPU* de 1 *MIPS* [3] . En appliquant cette règle aux machines parallèles actuelles, le plus «petit» calculateur du TOP 500 [4] requiert une bande passante de 2 To/s alors que le plus performant nécessiterait plus de 38 To/s. L'intérêt de ces propos n'est pas de remettre en cause la justesse de la loi proposée par Amdahl mais d'illustrer l'écart qu'il existe entre les performances des *CPUs* modernes et les débits fournis par les unités de stockage existant sur le marché. Écart rappelé dans [HP96] où il est indiqué que si la rapidité des processeurs augmente d'environ 35% à 50% et que les temps d'accés à la mémoire diminuent aux alentours de 60% par an depuis la dernière décennie, les performances apportées

[1] Technique visant à rendre transparente et performante l'utilisation, pour le calcul, de plusieurs ordinateurs et/ou super-calculateurs inter-connectés par un réseau.

[2] Nous utiliserons par la suite l'acronyme E/S pour désigner les termes d'entrées/sorties.

[3] *Million Instructions Per Second*.

[4] Liste des 500 plus puissantes plates-formes de calcul disponibles, novembre 2005, `http://www.top500.org`.

aux sous-systèmes d'E/S restent très marginales en atteignant difficilement 10%. Cet écart est par ailleurs amplifié par les modes d'accès parallèles mis en œuvre dans les applications scientifiques modernes, comportements qui affectent considérablement l'efficacité des systèmes de stockage sous-jacents.

Il est donc primordial de proposer des solutions matérielles et/ou logicielles complémentaires capables d'exploiter de manière efficace ces supports de sauvegarde et d'éliminer au maximum l'ensemble des différents goulets d'étranglement susceptibles d'intervenir au cours des exécutions.

Deux principaux axes de recherche ressortent des différentes approches proposées par la littérature. La première classe de solutions se compose des techniques d'analyse de code durant la phase de compilation. La création d'un graphe de flot de données à partir des instructions offre la possibilité de mettre en place diverses techniques d'optimisation et diminue, ainsi, l'effet indésiré de toute latence liée aux E/S. Toutefois, la complexité des codes scientifiques (irrégularité des problèmes, dépendance entre les instructions) ne permet pas dans de nombreux cas de retrouver ce graphe de flot de données de manière transparente, il est alors indispensable que le développeur indique par l'intermédiaire d'instructions explicites cet enchevêtrement d'opérations d'échange de données.

Le second groupe, celui auquel nous nous intéresserons, consiste à intervenir durant l'exécution du programme : soit au niveau des systèmes de partage de données en proposant des modèles adaptés aux nouvelles exigences, soit au niveau des routines d'accès en complétant l'interface usuelle `POSIX` (`open`, `close`, `read`, `write` ...) souvent inadaptée dans les applications haute performance [5] fortement dépendantes des unités de stockage.

Positionnement et contribution des travaux

Si l'avènement des architectures «inter-connectées» pour le calcul a permis de fournir des puissances d'analyse considérables à un «bon prix», il a également complexifié la mise en œuvre d'un système de partage de données au sein des architectures parallèles. Á l'opposé des systèmes de fichiers locaux utilisés par les super-calculateurs [6] , les systèmes de fichiers présents dans les grappes doivent proposer des méthodes d'accès distantes permettant d'exploiter un même fichier indépendamment de la localité où s'exécute l'opération. Ces solutions de partage distribuées sont communément évaluées par plusieurs facteurs : réactivité, cohérence, transparence d'accès, disponibilité ou encore possibilité au passage à l'échelle. En milieu *HPC*, les facteurs réactivité et passage à l'échelle nous intéressent particulièrement :

- La réactivité évoque la quantité de temps nécessaire pour satisfaire une requête (latence réseau, analyse de la demande et formulation de la réponse).
- Le passage à l'échelle s'attarde sur la capacité du système de fichiers à traiter un nombre plus ou moins important de clients.

Ces critères sont primordiaux dans notre cas puisqu'ils permettent de juger directement de la performance du système. Contenu du nombre croissant de nœuds qui compose les grappes et des systèmes de gestion de ressources associés qui permettent une utilisation optimale des

[5] «*High Performance Computing*», nous utiliserons par la suite l'acronyme *HPC*.

[6] Système composé de plusieurs processeurs au sein d'une même machine, généralement présenté sous l'acronyme *SMP* («*Symetric Multiple Processor*») ou *MPP* («*Massive Parallel Processor*»).

machines [FRS04], un plus grand nombre d'applications parallèles est généralement exécuté par rapport à une machine multi-processeur. Toutefois, l'exécution en parallèle d'un nombre important d'applications engendre une concurrence de plus en plus significative au niveau des supports de stockage.

De nombreuses solutions «dynamiques» ont été proposées depuis les quinze dernières années. Elles tentent de réduire les phénomènes de congestion afin d'améliorer les performances durant l'exécution des programmes.

Deux catégories, abordées de manière plus détaillée au cours du document, se distinguent : les systèmes de fichiers dits «parallèles» et les bibliothèques spécialisées dans les modes d'accès propres aux applications parallèles.

La première solution consiste à proposer un système complet capable de gérer les requêtes depuis l'application cliente aux zones de stockage physiques réparties sur l'architecture tout en tentant de répondre à un large nombre de critères (cohérence, disponibilité, tolérance au pannes, performance, passage à l'échelle ...). La grande diversité des aspects traités dans ces systèmes impose souvent certaines concessions d'un point de vue des performances.

La seconde possibilité se concentre sur l'élaboration de techniques visant à optimiser la manière dont les applications accèdent aux supports de stockage indépendamment du système de fichiers sous-jacent. La notion de «portabilité des performances» est généralement employée pour définir ces approches. Elles proposent, via des interfaces plus ou moins lourdes, divers algorithmes (agrégation, recouvrement, ré-ordonnancement, caches) permettant d'améliorer les performances. Toutefois, deux aspects importants de ce genre d'approches sont critiquables : dans un premier temps, leur utilisation requiert une compréhension rigoureuse de chacune des subtilités qu'elles proposent (création de types structurés, définition de masques d'accès, ...) afin de ne pas engendrer de surcoût inutile en plus d'augmenter considérablement le temps de développement. Cette prise en main parfois fastidieuse est un réel frein pour l'essor de ces technologies et un grand nombre d'applications parallèles continue à exploiter les interfaces standard, plus simples à appréhender. Le second point et non le moindre, porte sur le fait que les optimisations proposées ne prennent pas en compte les interactions émanant des applications concurrentes. De ce fait, les stratégies collectives généralement mises en œuvre s'avèrent inutiles et coûteuses lorsqu'elles entrent en conflit avec d'autres requêtes provenant d'applications concurrentes au niveau des serveurs de stockage.

Les travaux réalisés durant ces trois années, ont consisté à étudier dans quelle proportion l'utilisation des interfaces standard `POSIX` couplée à divers mécanismes de la thématique des E/S parallèles pouvait permettre d'améliorer l'accès aux données dans un environnement réparti multi-applicatif destiné au calcul intensif.

La contribution principale de cette thèse est double : d'une part, elle consiste en la proposition et l'étude d'une stratégie d'ordonnancement globale des E/S dans un contexte multi-applicatif haute-performance. D'autre part, elle a permis de mettre en œuvre un service générique d'ordonnancement des E/S permettant d'évaluer et de comparer plusieurs algorithmes. Ce service, intitulé *aIOLi*, est totalement transparent d'un point de vue des applications et de ce fait ne nécessite aucune interface complémentaire.

Ce nouveau système a été développé en tant qu'un module *Linux* (3300 lignes) susceptible d'être inter-connecté à une large gamme de systèmes d'E/S du noyau. Ce module est librement disponible à l'adresse `http://aioli.imag.fr`. Un «*patch*» fonctionnel permettant de faire interagir le service *aIOLi* et le système de fichiers `NFS` est également téléchargeable. Les diverses évaluations réalisées sur un serveur `NFS`, ont montré l'apport significa-

tif sur les performances de l'ensemble des applications. Un second «*patch*» a été développé afin de proposer le service au niveau de la VFS [7] . Toutefois, ce dernier développement n'a pas été assez évalué et même s'il est disponible, il reste à l'état expérimental.

Depuis fin 2005, le projet *aIOLi* est mené en étroite collaboration avec l'Institut ICIS de Czestochowa situé en Pologne.

Dans un premier temps, nous avons analysé la gestion des E/S au sein d'un unique nœud *SMP* sous le système *GNU/Linux*. Cette étude préliminaire, nous a permis de déterminer une des notions fondamentales de notre proposition : la «sérialisation» des accès. Comme nous le verrons par la suite, l'utilisation de ce concept permet de rétablir, à partir d'un mode d'accès parallèle souvent critique, un mode d'accès séquentiel nettement plus performant pour une application et ce en s'appuyant uniquement sur les routines `POSIX`. La bibliothèque qui a permis de valider ce concept n'est pas directement abordée au sein du mémoire puisqu'elle a consisté en une étape intermédiaire qui nous a permis de proposer le modèle global. Plusieurs rapports et publications qui la décrivent sont accessibles [LD05b, eYD05, LD05a]. Une description complète de ce système aurait compliqué la lecture du document sans pour autant constituer un réel apport.

La seconde phase des travaux qui a débuté courant mars 2005 à consister à proposer une solution à l'échelle d'une grappe. La définition d'un modèle permettant de déterminer l'ensemble des interactions présentes au sein d'une grappe nous a permis de définir deux politiques d'ordonnancement. Un de ces deux algorithmes a été particulièrement évalué et permet d'améliorer les performances pour chaque application tout en proposant une répartition «équitable» des temps d'accès à l'unité de stockage. La première implantation réalisée nous a montré la difficulté de mettre en œuvre une solution simple et non intrusive [LDV05]. La dernière étape, la plus conséquente, a consisté au développement et à l'évaluation d'un module noyau *Linux* indépendant et générique. Ce dernier développement a abouti sur la proposition *aIOLi*, un service générique de stratégie d'ordonnancement pour système d'E/S. C'est au sein de ce module couplé à un système de fichiers `NFS` que nous avons pu évaluer et affiner depuis l'été 2005 un de nos algorithmes. Les évaluations menées sur 96 nœuds de la grappe de Sophia-Antipolis de l'architecture Grid 5000 ont montré le réel apport de notre proposition [LDHS06, LHSD06, LHS06].

Plan du document

Méthodes d'accès locales, système de gestion de base de données, partage de ressources au sein d'un réseau d'entreprise ou encore échange d'informations à l'échelle planétaire via les nombreux systèmes pair à pair ou encore le web ; la problématique du partage des données a toujours suscité un engouement certain. La large diversité des travaux de recherche menés rend très complexe voire impossible la rédaction d'une section recouvrant l'ensemble des problèmes inhérent à cette thématique. Plusieurs ouvrages servent de référence et permettent au lecteur de se familiariser avec des concepts élémentaires [Tan01] ou des aspects plus spécifiques des systèmes de fichiers distribués [LS90].

L'état de l'art présenté dans le document se concentre sur les critères de performance des E/S au sein des applications s'exécutant sur les architectures parallèles distribuées. Les aspects

[7] *Virtual File System cf. 1.2.2.1*

de disponibilité, de sécurité, de résilience ou encore de tolérance aux pannes ne sont pas traités au sein de ce document.

Cette étude de l'existant se compose de trois chapitres. Dans le premier, nous décrivons les différentes entités matérielles puis logicielles qui interviennent dans la sauvegarde et l'extraction des données ; depuis l'unité fondamentale qu'est le disque dur en passant par la gestion des données au sein d'un système local. Cette première étape dans la lecture du document va permettre de mettre en évidence les nombreux points ayant un rapport direct avec les performances.

Le second chapitre présente des notions plus spécifiques à la problématique des E/S parallèles. Une caractérisation assez générale des modes d'accès propres aux applications s'exécutant sur des architectures dédiées au calcul intensif débute cette partie. Les stratégies d'optimisation les plus connues qui découlent des nombreux travaux corollaires sont décrites par la suite. Enfin, les principales solutions réalisées autour de ces approches sont présentées et analysées au sein du chapitre 3. Un bilan général présentant nos objectifs conclut cet état de l'art.

La seconde partie du document se consacre plus particulièrement aux travaux réalisés. Le chapitre 4 décrit une étude préliminaire des performances d'une application parallèle accédant de manière distribuée à un fichier. Cette étape montre plusieurs lacunes dans les méthodes habituelles d'accès aux données et permet de justifier l'intérêt de mettre en œuvre une stratégie d'ordonnancement à plus haut niveau.

Le chapitre 5 expose le modèle établi pour une gestion efficace et globale des E/S et présente l'algorithme d'ordonnancement par défaut de notre modèle. Dans le chapitre suivant (chapitre 6), nous abordons les difficultés de mettre en œuvre un service, transparent, de contrôle et de régulation des E/S pour l'ensemble des interactions émanant de l'architecture. Les étapes successives qui nous ont amené à proposer une implantation au niveau du noyau sont également traitées. L'ensemble des évaluations conduites pour valider la proposition *aIOLi* figure au chapitre 7. Le chapitre 8 établi un bilan général de la proposition implantée.

Lors de la conclusion, nous évoquons, les diverses perspectives entrouvertes autour de l'intégration de modules d'ordonnancement couplés à des gestionnaires de caches. Nous mentionnons également l'importance d'une prise en compte des besoins en terme d'E/S de chacune des applications lors de leur soumission par les gestionnaires de ressources au sein des architectures de type grappe ou grappe de grappes.

I

Contexte d'étude

C'est presque tout que de savoir lire.
Emile-Auguste Chartier

INFORMATIQUE ET CONTRAINTES DE STOCKAGE 1

Dans ce chapitre, nous introduisons la problèmatique du stockage au sein des architectures informatiques. La thématique recouvrant un spectre très large [1] [Sto98] [HK04] , nous avons choisi d'aborder cette problématique en proposant une vision du stockage axée autour des performances.

Une bonne compréhension des échanges et des optimisations depuis l'unité de stockage physique jusqu'aux applications situées en espace utilisateur va permettre au lecteur de mieux appréhender les problèmes de manipulation des données en parallèle, qui seront traités dans le chapitre suivant.

Ce chapitre se décompose en deux sous-parties. La première développe les principales caractéristiques ainsi que le fonctionnement d'un disque dur. Cette description a pour principaux buts :

- d'une part, de présenter les évolutions qui ont été réalisées sur cette technologie et les impacts qu'elles ont eu sur les méthodes d'utilisation ;
- et d'autre part, d'aborder et de comprendre les contraintes auxquelles doivent faire face les couches de gestion des E/S des systèmes d'exploitation.

Ces différentes structures logicielles sont décrites dans la seconde partie. Une analyse de la pile des E/S du système *GNU/Linux* 2.6 sera établie. Même, si comme nous le verrons, les concepts présentés tout au long de ce mémoire peuvent être appliqués à différents systèmes d'exploitation, nous avons choisi de concentrer nos travaux sur le système d'exploitation *GNU/Linux* puiqu'il reste la référence dans les architectures dédiées au calcul intensif en étant utilisé par plus de 70% des machines du TOP500.

[1] A titre anécdotique, une archive centralisant une grande partie des travaux a été mise en place au sein du département informatique de Dartmouth (`http://www.cs.dartmouth.edu/pario`)

1.1. Unités de stockage persistantes

La majeure partie des ordinateurs exploitent des supports persistants afin de stocker et/ou récupérer des informations (CD/DVD, bandes magnétiques, mémoires non-volatiles ou encore disques durs). Les principaux avantages de ces supports par rapport aux supports volatils (comme la mémoire vive) sont, d'une part, leur capacité de stockage généralement plus importante et, d'autre part, leur coût de production, à espace de stockage équivalent, beaucoup plus faible. Malheureusement en plus de proposer pour certains d'entre eux des méthodes d'accès réduites [2] , le temps d'accès moyen à une zone de données particulière est souvent plus conséquent sur ces unités de stockage (cf. figure 1.1).

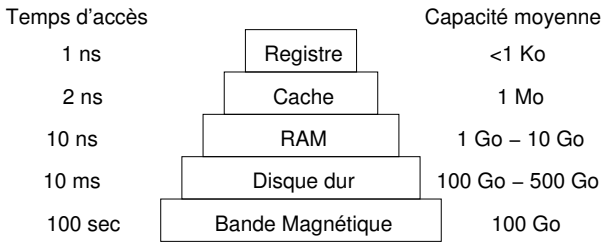


FIG. 1.1 – Temps d'accès et capacité des différents supports de stockage

Les disques durs offrent le meilleur compromis en termes de capacité, de performance et de modes d'accès, ce dernier comparable à celui disponible dans les mémoires volatiles. Toutefois, même si la conception de ce type de support a nettement évolué depuis le premier modèle en 1956 (figure 1.2) ; la micro-mécanique exploitée en interne engendre de nombreuses restrictions, en particulier sur les temps de positionnement lors d'accès purement aléatoires. Une description des caractéristiques fondamentales des disques durs ainsi que les évolutions majeures de cette industrie va nous permettre de mieux comprendre ces aspects et les raisons de ces limitations.

1.1.1. Caractéristiques générales

Représenté à la figure 1.3, un disque dur est une unité de stockage magnétique [Tan01]. Il est composé de plusieurs plateaux fixés à un même axe tournant à une vitesse constante (de 7200 tours/min, 10000 tours/min et pour les plus rapides de 15000 tours/min). L'information est stockée sur les différentes surfaces par cercles concentriques appelés pistes. Chaque piste est divisée en secteurs. Les données sont inscrites et récupérées par secteur par l'intermédiaire de têtes de lecture/écriture. Un grand nombre d'améliorations a été apporté depuis l'invention de la première unité en 1956, [JC03, RJTM03, GH03, HBP+81] : l'IBM RAMAC offrait une capacité de stockage de 4,4 Mo répartis sur 50 plateaux de 24 pouces tournant à 1200 tours/min ce qui permettait d'atteindre à l'époque un débit d'environ 8 Ko/s pour un temps d'accès moyen de 6 secondes. Un unique bras composé de deux têtes de lecture/écriture parcourait les deux

[2] Les lecteurs de bandes magnétiques n'offrent qu'un mode d'accès séquentiel par exemple.

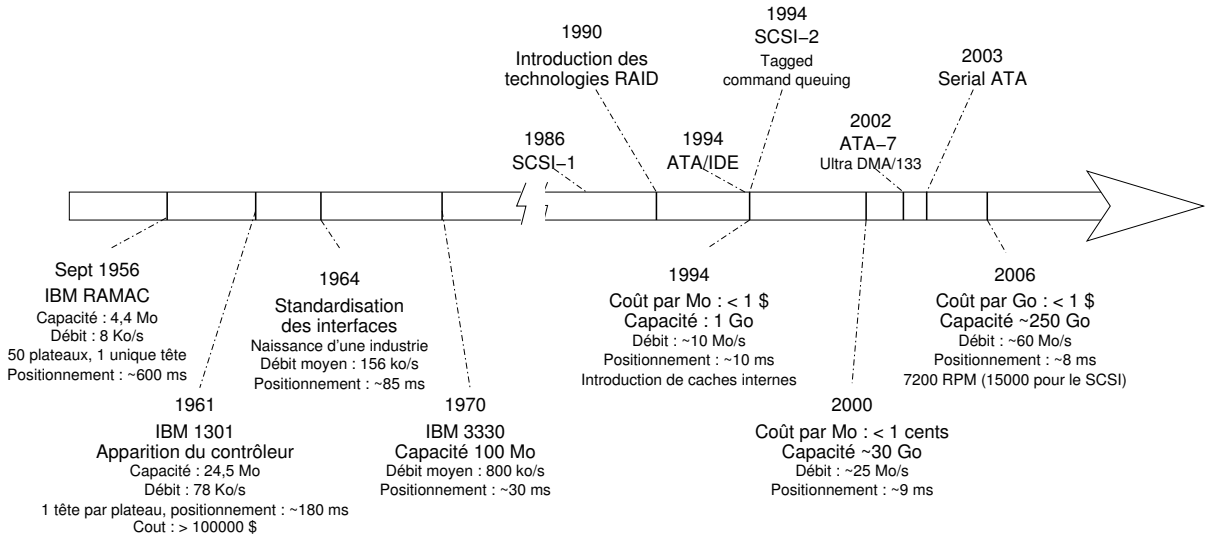


FIG. 1.2 – Évolution des disques durs depuis 50 ans.

surfaces d'un plateau ; le temps entre deux accès pouvait ainsi devenir rapidement significatif lorsqu'il était nécessaire de repositionner le bras sur un plateau distinct.

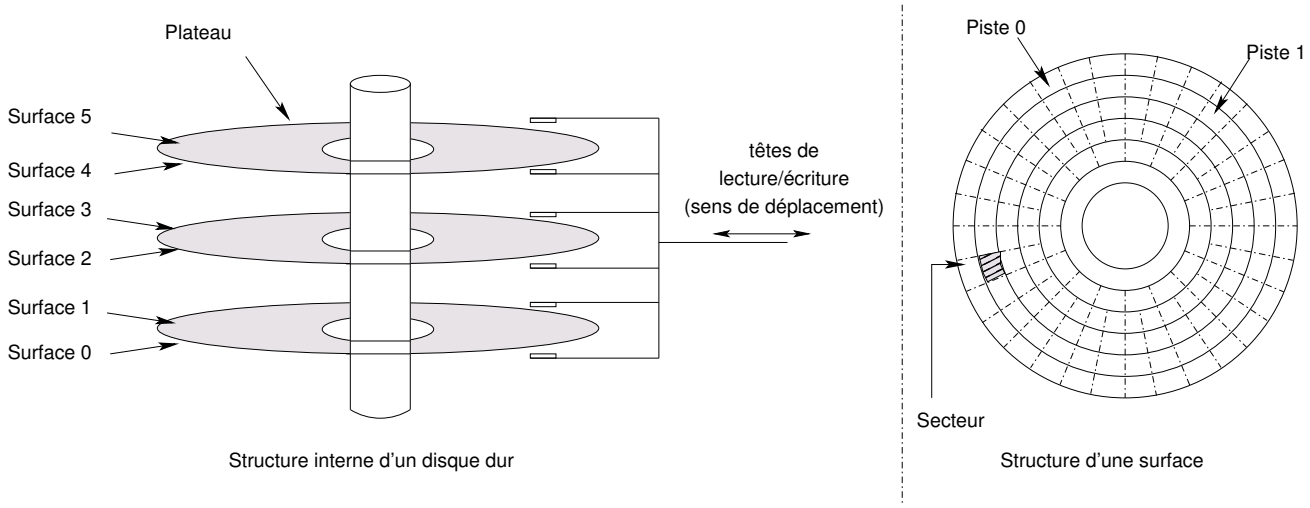


FIG. 1.3 – Composition interne d'un disque dur

Rapidement, une tête de lecture/écriture a été associée à chacune des surfaces. Cette première amélioration a permis de diminuer considérablement le temps moyen pour accéder à une zone spécifique du disque. Chaque tête est reliée à un même bras mécanique qui se déplace entre la périphérie et le centre des plateaux. La combinaison du déplacement transversal

du bras et de la rotation des plateaux permet aux têtes de parcourir la totalité de l'espace de stockage. L'ensemble des pistes accessibles à une position donnée du bras s'appelle un cylindre. Aujourd'hui la taille d'un secteur est de 512 octets et devrait à terme croître vers 4 Ko.

1.1.2. Coût et Capacité de stockage

L'évolution technologique majeure a été réalisée au niveau du facteur de densité de stockage (multiplié par 35 millions en 50 ans). Cette optimisation a permis d'augmenter considérablement les capacités tout en réduisant la taille des unités [3] .

A la manière de la loi de Moore pour la puissance CPU, la capacité des unités de stockage suit la loi de Kryder [4] en doublant tous les 13 mois. En 1998, année où l'on commémorait le centenaire de l'enregistrement magnétique (inventé par le Danois Valdemar Poulsen), IBM proposa le premier disque dur de 25 Go, capacité présentée à l'époque comme disproportionnée aux besoins réels des particuliers (5 ans plus tard, 80 Go étaient considérés comme une taille à peine suffisante). En 2002, plus de 200 millions de disques ont été produits pour une capacité cumulée proche de 1 Exa-octets [5] . En 50 ans, la capacité d'une unité de stockage a été multipliée par un facteur supérieur à 100000 pour atteindre aujourd'hui 500 Go répartis sur 3 plateaux de 3.5 pouces [GH03].

Parallèlement à la forte progression des densités de stockage et dans une logique de forte industrialisation, le coût de production des supports a été divisé par 5 depuis 1980. En 1996, il était d'ores et déjà économiquement plus rentable de stocker des informations à quantité égale sur ce type de support que sur du papier ou du film. Toutefois, le prix de revient d'une unité de stockage est évidemment dépendant des performances qu'il propose.

Deux catégories se distinguent selon leur interface d'exploitation [WNH01, ADR03] :

- Les disques de type *IDE* (*Integrated Device/Drive Electronics*) sont les unités les plus répandues (plus de 85% du marché). Par leur prix de revient meilleur, ils sont principalement destinés à l'informatique de grande distribution (PCs) et sont reconnus parfois à tort, comme étant moins performants.
- Les disques *SCSI* (*Small Computer System Interface*) sont considérés comme les supports haut de gamme et équipent les serveurs ou encore les super-calculateurs. Leur prix plus élevé est induit d'une part par une mécanique plus fiable et généralement plus performante et d'autre part par une interface de communication plus riche. Les commandes transmises au périphérique peuvent être plus ou moins complexes et seront éventuellement décomposées en sous-tâches plus simples au niveau du disque.

D'un point de vue global, les éléments liés à l'exploitation des données sur ces unités (micro-contrôleur, caches internes, micro-code, ...) représentent une part croissante dans le coût de production par rapport aux pièces de micro-mécanique et atteignent d'ores et déjà plus de 2/3 du prix. L'utilisation d'une logique interne de plus en plus complexe a pour but d'améliorer les performances régies en grande partie par les contraintes imposées par les composants mirco-mécaniques.

[3] A titre anécdotique, Seagate a présenté un nouveau disque dur de 1 pouce (40 * 30 * 5 mm) offrant une capacité de 12 Go début 2006

[4] De l'ingénieur Mark Kryder de la société Seagate,
`http://en.wikipedia.org/wiki/Kryder%27s_law.`

[5] *kilo,mega,giga,tera,peta* et *exa* (10^{18}).

1.1.3. Performances et temps d'accès

Les améliorations en termes de performances (débit, temps d'accès) sont nettement moins significatives que les progrès réalisés en termes d'espace de stockage. Les débits internes sont soumis à des contraintes physiques fondées sur la vitesse de rotation, la densité de stockage et le diamètre des plateaux. Ils atteignent aujourd'hui quasiment 100 Mo/s avec un facteur de croissance de 40% par an.

Toutefois, ces performances «brutes» ne sont pas significatives puisqu'elles ne correspondent pas aux débits effectifs lors de l'exploitation des supports. Pour y parvenir, les accès doivent être séquentiels et les données contiguës sur l'unité de stockage de manière à éviter tout mouvement de bras. En effet, le positionnement des têtes de lecture/écriture sur une zone spécifique nécessite une opération mécanique coûteuse. Certes, les améliorations technologiques ont permis de réduire ce temps de 50 ms dans les années 1970 à une moyenne d'environ 8 ms pour les disques *IDE* actuels et 5 ms pour les unités plus spécialisées comme le *SCSI* mais cette opération reste un point critique dans les performances. Elle est, aujourd'hui, l'opération qui nécessite le plus de temps dans les ordinateurs modernes.

Le temps d'accès à une zone spécifique est donné par la formule suivante :

$$tps\_accès = tps\_de\_positionnement + latence$$

Le temps de positionnement correspond au temps nécessaire au déplacement des têtes d'un cylindre vers un autre : à titre indicatif, le déplacement d'un cylindre vers le suivant prend environ 1 ms pour atteindre quasiment 10 ms vers un cylindre quelconque. La latence correspond au temps nécessaire pour que la zone adéquate se retrouve sous la tête de lecture/écriture. Cette valeur est liée à la vitesse de rotation (de l'ordre de 4.5 ms pour les unités tournant à 72OO tours/min). De ce fait et même si le terme débit est usuellement utilisé, la valeur généralement fournie par les constructeurs correspond au nombre moyen de requête d'E/S par seconde pour une granularité fixée. Ce nombre étant directement corrélé au temps d'accès, il est donc possible de l'améliorer en réduisant la taille des plateaux d'une part (le bras ayant une surface plus courte à balayer, le temps de positionnement est moins important) et d'autre part en faisant tourner ces derniers plus vite (diminuant ainsi la latence). Toutefois, cette approche généralement employée dans les disques *SCSI*, nécessite des éléments mécaniques et électroniques plus onéreux afin de supporter les différentes perturbations physiques (chaleur, électromagnétique, vibration, ...) et donc une augmentation significative du coût de production globale.

Une approche complémentaire consiste à intégrer aux périphériques un jeu d'instruction plus complexe ainsi que différents tampons mémoire permettant de ré-ordonner des requêtes, d'exploiter des techniques de pré-chargement ou encore des stratégies dites d'écriture retardée. Une bonne approximation des performances doit donc prendre en compte le débit interne, le temps d'accès moyen et la taille des caches internes. Nous abordons ces aspects dans les paragraphes suivants.

1.1.4. Logique interne

Plusieurs techniques «logicielles» ont été proposées afin de réduire au maximum le surcoût engendré par le temps nécessaire au repositionnement des têtes.

Jusqu'en 1994, les unités de sauvegarde pouvait recevoir au sein du contrôleur une requête à la fois. Chaque accès se déroulait ainsi de la manière suivante [6] :

- La première étape consiste à déterminer l'emplacement sur l'unité de stockage des informations souhaitées. Cette opération est réalisée par les différentes couches qui séparent l'application logicielle de l'unité de stockage physique et peut parfois entraîner plusieurs traductions (couches internes au système d'exploitation, *BIOS* ou pilote spécifique à l'unité de stockage, cf. section 1.2.2). Pour une requête logique (au sein d'un fichier), un ou plusieurs accès physiques sont nécessaires. Ils sont exprimés en termes de cylindre, de tête et de secteur. Généralement, les secteurs ne sont pas lus ou écrits individuellement, ils sont regroupés en blocs continus permettant de limiter les déplacements. Il serait en effet, beaucoup trop coûteux de récupérer ou de sauvegarder un fichier de quelques Mo par bloc de 512 octets.
- Lorsque la localisation des données sur le disque est déterminée la première sous requête physique est transmise à l'unité de stockage. Le contrôleur du disque vérifie si la zone demandée est disponible au sein de ses tampons internes. Cette zone mémoire,. appelée *buffer-cache*, contient généralement les derniers blocs lus ou inscrits. Dans le cas positif, les informations sont directement renvoyées : aucune opération mécanique n'est alors réalisée. Dans le cas contraire, le contrôleur interprète l'adresse afin de déterminer le déplacement requis pour positionner le bras sur le cylindre adéquat. La lecture (ou l'écriture) débutera lorsque la zone appropriée se trouvera sous la tête. Dans le cas d'une lecture, les données sont copiées au sein du *buffer-cache* avant d'être renvoyées par l'interface (*IDE/SCSI*) au système ayant émis la demande. La requête suivante peut alors être traitée.

Mécanismes de *cache*

L'utilisation de *buffer-cache* a été une des premières techniques mises en œuvre afin de réduire les déplacements. Il consiste en une zone mémoire volatile associée, physiquement, à l'unité de stockage. Cette zone est exploitée afin de garder de manière temporaire les dernières données qui ont été manipulées. L'espace étant limité, une politique de gestion permettant par exemple de déterminer quels sont les blocs qui doivent rester en mémoire et quels sont ceux qui peuvent être supprimés est requise. Les premières stratégies consistaient juste à garder en mémoire les derniers blocs lus ou écrits (*First In First Out*).

Par la suite, plusieurs améliorations ont été apportées :

- Taille de bloc dynamique
  Les *buffer-caches* initiaux étaient divisés en segment de taille fixe. Lors d'une lecture sur une zone ne nécessitant que l'accès à une sous partie d'un segment, la totalité du segment était récupérée et stockée conduisant ainsi à une utilisation non optimale de l'espace disponible. La majeure partie des nouveaux périphériques intègrent des mécanismes permettant de définir des tailles ainsi qu'un nombre dynamique de segments et fournissent ainsi une meilleure utilisation des espaces.
- Pré-chargement :
  En s'appuyant sur les derniers accès réalisés, le contrôleur va pré-charger au sein du *buffer-cache* des segments susceptibles d'être demandés lors des prochaines requêtes. Cette

[6] Les corrections/vérifications d'erreurs et les autres techniques visant à rendre plus fiables les supports ne sont pas incluses dans la description.

technique permet de recouvrir le temps nécessaire au positionnement sur les zones adéquates avec les temps de traitement du système entre deux requêtes : la première requête engendre éventuellement un positionnement coûteux, la réponse est transmise au système ; en attendant que la requête suivante soit déposée, le contrôleur pré-charge les données suivantes. La seconde requête arrive, elle est alors immédiatement servie par le *buffer-cache*. Cette approche améliore significativement les performances dans le cas d'accès purement séquentiels.

– Interface de contrôle :
Les dernières interfaces d'exploitation [7] des unités de stockage ont été enrichies par un ensemble de routines ayant pour but le paramétrage du *buffer-cache* (activation/arrêt des techniques de pré-chargement, taille des segments, ...) ou tout simplement son arrêt. En effet, à la différence des comportements séquentiels, il est parfois plus intéressant de désactiver l'utilisation des tampons internes lorsque la majeure partie des accès sont aléatoires et concernent différentes parties du disque. Si aucune requête n'a accédé récemment aux données, elles ne pourront être présentes au sein du *buffer-cache*. Un surcoût est alors associé à chaque fois que le contrôleur vérifie si les données apparaissent ou non dans les tampons internes. Dans ce cas précis, il peut être plus avantageux de court-circuiter le *buffer-cache*.

S'il est vrai que les objectifs fondamentaux des techniques de cache restent les mêmes quelque soit le type de requêtes, il est important de noter qu'une opération d'écriture se différencie d'une lecture par le fait qu'elle engendre une modification physique du support. Ainsi, les techniques mises en œuvre sont différentes. Pour les écritures, deux modes sont généralement proposés :

– Le premier (*write-through*) consiste à exécuter l'écriture sur le support physique en plus de la copier au sein du *buffer-cache* en vue d'une potentielle lecture. Certes, l'impact du au positionnement des têtes peut rapidement être significatif mais en cas de panne, tout écriture acquittée par le contrôleur au système d'exploitation a réellement été inscrite sur l'unité. Cette approche est généralement employée afin de garantir un état cohérent des données par rapport au système d'exploitation.
– Dans la second (*write-behind*), lorsque le contrôleur reçoit une demande d'écriture, il utilise le *buffer-cache* comme zone temporaire : les données y sont copiées et un acquittement est immédiatement transmis au système d'exploitation. La réalisation effective de l'écriture sera réalisé par la suite par le contrôleur. Cette approche améliore considérablement les performances en écriture. Cependant, puisque le *buffer-cache* est une mémoire volatile, toutes les requêtes de type écriture en attente sont perdues en cas de panne. Il est en effet impossible au système d'exploitation de connaître les requêtes qui ont été détruites puisqu'elles ont été précédemment acquittées.

Ré-ordonnancement des requêtes

En 1994, le standard *SCSI-2* a permis d'introduire le concept de *Queuing*. L'approche repose sur le fait que cette nouvelle interface permet de transmettre plusieurs requêtes au micro-contrôleur. Ainsi, afin d'éviter des repositionnements coûteux pour les accès ne pouvant être satisfaits par le cache, le micro-contrôleur va pouvoir redéfinir un ordre de traitement pour les requêtes en attente.

[7] *API, Application Program Interface.*

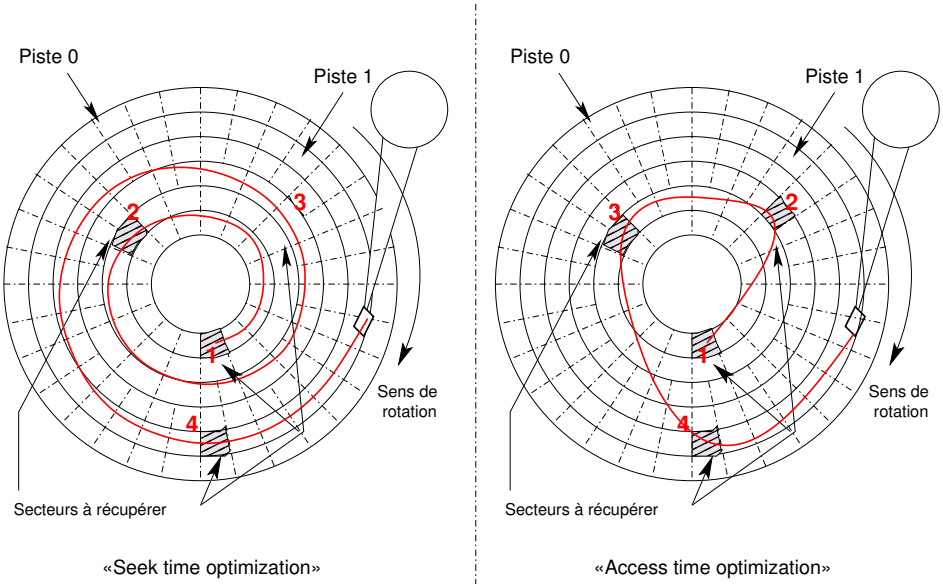


FIG. 1.4 – Algorithme d'optimisation des temps de positionnement

Plusieurs algorithmes existent [Tho04] ; cependant d'une manière générale, ils appartiennent à l'une de ces 3 catégories :

- *First In First Out*, les requêtes sont traitées selon leur ordre d'arrivée.
- *Seek Time Optimization*, les requêtes sont analysées puis réordonnées selon le numéro de cylindre correspondant et le positionnement actuel du bras.
- *Access Time Optimization*, les requêtes sont analysées puis réordonnées selon le numéro de cylindre et le positionnement au sein de la piste. En effet, les temps de latence ne sont pas pris en compte dans la classe d'algorithme précédente. Deux accès peuvent ainsi être situées sur deux cylindres consécutifs mais à l'opposé au niveau des pistes pendant qu'un troisième accès pourrait être situé sur un cylindre plus éloigné mais plus proche de la première requête d'un point de vue piste. Les algorithmes les plus avancés analyseront pour chaque accès le temps nécessaire au déplacement du bras mais également le temps nécessaire pour que les données se retrouvent sous la tête de lecture/écriture. La mise en œuvre d'un tel algorithme est nettement plus complexe et requiert de la part du micro-contrôleur une bonne appréciation de ses paramètres physiques. En effet, si le délai prévu pour déplacer le bras sur le prochain cylindre est plus long que prévu, la tête de lecture/écriture va manquer le secteur et devra pas conséquent attendre la rotation complète du plateau afin que la zone se représente à nouveau sous la tête.

Les deux dernières classes sont illustrées par la figure 1.4. Les stratégies d'ordonnancement internes aux disques sont connues sont le nom de *command queuing*. La longueur de la queue de réception est appelée «profondeur de file». Dans la pratique, la profondeur normale de la file est le plus souvent de 64 commandes.

Depuis 1998, la norme *ATA-4* [8] a permis d'intégrer de telles fonctionnalités au sein des disques *IDEs*.

[8] La norme ATA, *Advanced Technology Attachment* est le standard pour les disques *IDEs* depuis 1994.

1.1.5. L'effet *ZCAV*

Depuis le début des années 1990, la plupart des disques exploitent une technique nommée *ZCAV* [9] . Ce mécanisme est employé afin d'exploiter l'espace disponible plus conséquent sur les pistes externes que sur les pistes internes aux plateaux, et sauvegarder, ainsi, un plus grand nombre de secteurs. Le ratio généralement mis en œuvre est de 3 pour 2 et peut aller sur certaines unités à 2 pour 1. En une révolution, la quantité de données lue ou écrite peut varier par un facteur jusqu'à 2 fois supérieur selon les cylindres accédés. Ainsi, le taux de transfert entre le disque et les tampons temporaires est différent selon les secteurs manipulés.

Si le rapport entre les pistes internes et celles externes est de 2/3, la récupération d'un fichier situé sur les pistes externes prendra 2/3 de temps de moins que s'il etait situé sur les pistes internes. Il est de même pour les déplacements puisque chacune des pistes à l'extérieur des plateaux contient plus de secteurs.

Cet effet bien connu [Van97] [ES03] est cependant rarement mentionné. Dans le cadre de travaux visant à évaluer les performances des systèmes d'E/S, il est primordial d'en tenir compte.

1.1.6. *Redundant Array of Independent / Inexpensive Disks*

Proposé à la fin des années 80 [PGK88], le concept *RAID* consiste à offrir un niveau d'abstraction permettant de regrouper plusieurs disques soit pour assurer un plus haut niveau de fiabilité (redondance des informations) soit pour augmenter les performances (répartition des informations) ou les deux. La nouvelle structure reste complètement invisible à l'utilisateur de l'ordinateur. En effet, quel que soit le nombre de disques physiques utilisés pour construire le *RAID*, l'utilisateur de la machine ne verra jamais qu'un seul grand disque logique, auquel il accédera de façon tout à fait habituelle.

Cette abstraction est fournie soit par des contrôleurs matériels spécifiques soit par une abstraction logicielle. Dans le cas de contrôleurs matériels, une carte ou un composant est dédié à la gestion des opérations. Ces cartes additionnelles offrent la possibilité de reconstruire de manière transparente les disques défaillants, de changer les propriétés ou d'étendre la capacité des unités *RAID*.

Plusieurs combinaisons ont été proposées selon les objectifs attendus : performance, fiablité et coût associé. Elles dérivent toutes de 8 assemblages principaux numérotés de 0 à 7 [CLG+94]. Nous présentons ici, les plus utilisées :

- Le ***RAID* 0** a été proposé afin d'améliorer les performances. Il consiste à répartir les données sur plusieurs unités de manière à paralléliser les accès sur les données [SGM86][10]. Les fichiers sont découpés en blocs (*chunk*) de taille fixe répartis sur les différentes unités. Cette stratégie permet de décupler en théorie, les débits. Cependant, le gain réel dépend de la corrélation entre la taille de découpage et les modes d'accès.
- Le ***RAID* 1** répond à des critères de fiabilité. Chaque donnée est dupliquée sur la totalité des disques qui compose la structure. Lors de la défaillance de l'un des disques, le contrôleur RAID (matériel ou logiciel) désactive instantanément le disque incriminé et bascule dans un mode dégradé où les traitements continuent à s'effectuer mais où la fiabilité n'est potentiellement plus assurée. Il est alors nécessaire de changer l'unité endommagée et de

[9] *Zone Constant Angular Velocity*, appelée également *Zone Bit Recording*.
[10] *data striping*

synchroniser la nouvelle unité avec les autres disques afin de rebasculer dans le mode normal. Cette combinaison est relativement coûteuse puisqu'elle nécessite deux fois la taille effectivement utilisable.

- Les ***RAID* 3 et 4** utilisent, en combinaison avec une structure *RAID* 0, un disque supplémentaire afin d'y sauvegarder un bit de parité soit par octet (*RAID* 3) soit par bloc *RAID* 4. Si un des disques du *RAID* 0 tombe en panne, il est possible de reconstruire l'information. Cependant si le disque de parité subit une défaillance. La fiabilité n'est plus assurée et il faut donc intervenir afin de changer l'unité et recalculer l'ensemble des parités.
- Le ***RAID* 5** consiste à répartir le bit de parité sur l'ensemble des disques composant la structure (corrigant ainsi le principal défaut des *RAID* 3 et 4).

D'un point de vue pratique les combinaisons du *RAID* 1+0 et du *RAID* 5+0 sont celles les plus déployées dans les centres de calculs. Elles assurent performance et fiabilité :

- Dans le ***RAID* 1+0**, les unités sont regroupées deux à deux afin de fournir du *RAID* 1. Par la suite, le *RAID* 0 est appliqué entre les groupes de deux disques. La probabilité de perdre des données est réduite, il faut que deux unités du même groupe tombe en panne pour que l'architecture ne soit plus opérationnelle.
- Le ***RAID* 5+0** repose sur la même idée, les unités sont groupées par trois disques en *RAID* 5 et les groupes sont montés entre eux en *RAID* 0.

Un paramètre à ne pas oublier lors de la mise en place d'une structure *RAID* est le nombre de cartes d'interface vers les disques (cartes IDE ou SCSI). En effet, si le nombre de disques est important, la carte d'interface à laquelle est connectée la structure deviendra vite un goulet d'étranglement lors des transfert de données entre le système d'exploitation et l'architecture de stockage.

1.1.7. Bilan

Les progrès significatifs réalisés sur la densité de stockage ont permis de proposer aujourd'hui des disques d'une capacité supérieure à 500 Go. Cette évolution couplée à la baisse des coûts de production a joué un rôle majeur dans le développement de nouveaux systèmes de résolution comme la génomique ou la climatologie ainsi que dans les nouveaux modes d'utilisations des supports de stockage.

Les applications scientifiques ayant accès à une plus grande puissance de calcul, s'appuient sur une quantité d'information de plus en plus significative. Les données à analyser deviennent de plus en plus conséquentes.

Ainsi, les besoins en stockage des centres de calculs ne cessent de croître [MBR03] (cf. tableau 1.1).

Cependant, si les contraintes d'espace semblent temporairement réglées, les aspects liés aux performances ont été délaissés. Les débits et le temps d'accès moyen aux informations ont par exemple peu évolué depuis ces quinzes dernières années. En 2006, les débits effectifs moyens d'une unité de stockage gravitent entre 45 Mo/s et 65 Mo/s. Pourtant ces aspects sont primordiaux pour les nouvelles applications. A titre d'exemple, l'*IN2P3* [11] met actuellement en place des systèmes capables d'enregistrer les résultats émanant des applications de la physique du nucléaire effectuées sur l'accélérateur du CERN [12] en Suisse. Lors de chaque expérience,

[11] Institut National de Physique Nucléaire et de Physique des Particules.
[12] *LHC, Large Hadron Collider*, `http://public.web.cern.ch/`

Année	Capacité	Architecture
1990	50 à 100Go	non-renseigné
1994	1 To	Serveur de stockage à base de *RAID* 5
1997	75 To	ASCI Mountain Blue - 48 noeuds de 128 processeurs - SGI
2002	700 To	ASCI Q
2005	3 Po	ASCI Purple - Lustre - 10000 processeurs - 50 To RAM
2006	6 Po	NASA RDS

TAB. 1.1 – Évolution des besoins de stockage pour les centres de calcul

ces systèmes devront être capable de supporter un débit soutenu de plus de 1 Go par seconde pour une totalité de plus de 5 Po par an.

Les solutions actuelles consistent à exploiter l'approche *RAID* afin d'offrir de tels débits. Cependant, les performances proposées par ces structures dépendent énormément des modes d'accès utilisés par les applications.

Avant de rentrer dans le vif du sujet et d'étudier les modes d'accès propres aux applications parallèles, nous allons tenter de comprendre la manière dont le système d'exploitation interagit avec les unités de stockage. Pour cela, nous allons décrire la pile des E/S en nous appuyant sur le système *GNU/Linux*. De manière similaire à la présentation du disque, nous allons nous intéresser aux caractéristiques propres à la performance et aborder les différents mécanismes logiciels généralement mis en place pour exploiter de manière efficace les unités de stockage sous-jacentes.

1.2. Système d'exploitation et Entrées/Sorties

Dans cette section, nous présenterons la gestion des E/S au sein d'un système d'exploitation. Le but est de mettre en évidence plusieurs concepts sur lesquels nous allons nous appuyer par la suite. Dans un premier temps, nous allons donner une description très succincte de ce qu'est un système de fichiers. Cette vision simplifée va nous permettre de réaliser quelques rappels de terminologie. Dans un second temps, nous nous intéressons de manière plus détaillée à l'analyse de la pile des E/S au sein du système d'exploitaion *GNU/Linux* et principalement aux différentes techniques mises en place afin d'améliorer les performances.

1.2.1. Préambule

Le travail d'un gestionnaire de fichiers consiste à classer les programmes et les données pour les rendre accessibles à la demande. L'ensemble de la littérature abordant ce sujet s'appuie sur une terminologie spécifique [Leb02] permettant de caractériser les diverses notions et techniques employées.

Une proriété fondamentale dans la gestion des données concerne la transparence d'accès. Un usager qui souhaite manipuler des données précises, se réfère à un fichier par un nom textuel. Cette information utilisée pour accéder à «l'information» est communément appelé méta-information. La plupart des personnes est généralement bien habituée à manipuler ce

type d'informations (date, taille du fichier, propriétaire, droit ...) sans se douter qu'elles sont primordiales pour le bon fonctionnement du système de fichiers. En effet en complément aux méta-informations «utilisateurs» sont associées des méta-informations de plus bas niveau qui permettent de localiser les données inhérentes au fichier sur les diverses unités de stockage.

Ainsi, un système de fichiers peut être vu comme étant composé de deux entités : un service de désignation et un service de gestion des données. Le premier crée et gère les répertoires (ajout, destruction de fichiers) alors que le second prend en charge les opérations sur les fichiers, comme la lecture, l'écriture ou l'ajout.

1.2.2. Pile des E/S au sein d'un système d'exploitation

De manière analogue au modèle *OSI* [13] qui sert de référence aux mises en œuvre des piles réseaux, la gestion des E/S dans les systèmes d'exploitation repose sur un modèle à base de couches. Chaque couche enrichit les couches inférieures en proposant de nouvelles fonctionnalités. Par défaut, les systèmes `UNIX`-*like* exploitent au moins 4 couches :

Les deux couches de plus bas niveau sont extérieures au système d'exploitation et peuvent être vues comme un seul et même niveau. En effet, la couche la plus basse, la couche «physique», n'est que très rarement accessible. Elle correspond au micro-code de chacune des unités qui peut être spécifique selon les fabricants ou les modèles. La couche vue par le système d'exploitation est la couche «logique». Cette couche permet d'abstraire les différentes unités de stockage autour d'un standard commun (*IDE* ou *SCSI*). Elle est nécessaire à l'interconnexion et au dialogue entre les contrôleurs disques et les systèmes d'exploitation. Les contrôleurs *RAID* matériels introduits précédemment se situent également à ce niveau.

Les deux couches internes au système d'exploitation sont la couche «bloc» et la couche «système de fichiers». Chacune se décompose en plusieurs sous modules que nous allons traiter dans la suite du document. D'un point de vue général, la couche «bloc» offre une vision uniforme de l'espace de stockage et permet de faire correspondre les données situées en mémoire aux secteurs des différentes unités de stockage. La couche «système de fichiers» fournit la couche utilisable par les applictions. C'est elle qui fait la correspondance entre la traditionnelle arborescence dossier/fichier et la représentation par bloc utilisée par la majorité des systèmes de fichiers.

Une troisième couche, le gestionnaire de *caches*, est traditionnellement placé entre la couche «système de fichier» et la couche «bloc». A la manière des *caches* internes aux disques, elle a pour but de minimiser les interactions avec les couches inférieures et donc les opérations coûteuses sur les disques. Par exemple, un fichier temporaire d'une durée de vie très brève n'a pas d'intérêt à être propagé vers le disque. Cette couche, qui d'un point de vue conceptuel reste facultative, permet d'améliorer considérablement les performances.

La figure 1.5 représente une vue simplifié des différentes couches présentes au sein du noyau *Linux* [Lov05].

Afin de faciliter la lecture du document, nous avons choisi de décrire les différents composants intervenant dans la gestion des E/S depuis les applications vers les unités de stockage. Nous pensons qu'il sera plus facile pour le lecteur d'apréhender les différentes couches en partant d'un niveau généralement connu qui consiste à lire ou écrire au sein d'un fichier pour terminer par la réprésentation de ces données sur un disque.

[13] Open Systems Interconnection

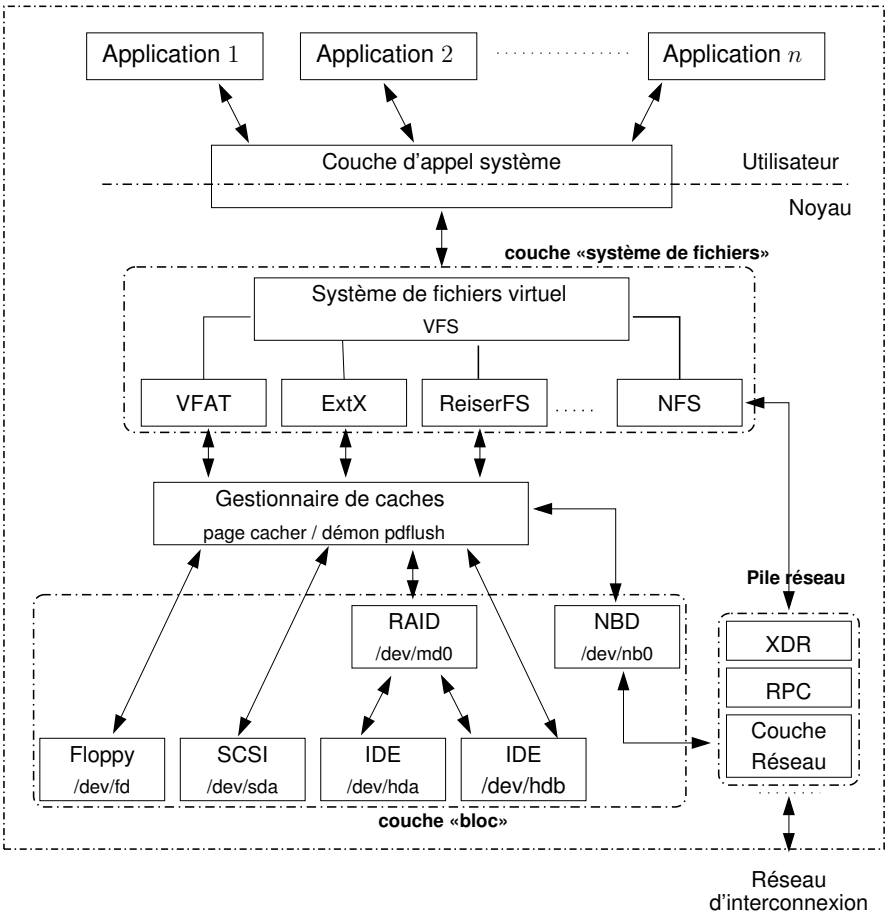


FIG. 1.5 – Pile des E/S au sein d'un noyau *Linux*

La couche «système de fichiers»

Les applications manipulent les données par l'utilisation de différentes bibliothèques permettant d'interagir directement avec différents niveaux de la pile des E/S.

La bibliothèque standard de l'interface `POSIX` (`open`, `read()`, `write()`, `close`, ...) traduit les requêtes applicatives en appels système. Ces accès «noyau» sont transmis au système de fichiers virtuel. Le rôle de cette couche consiste à fournir un accès transparent pour les applications aux différents systèmes de fichiers présents sur la machine.

Ainsi, lorsqu'un appel système transmet une requête à la *VFS* [14] , elle la relaye vers le système de fichiers qui en a la charge. Cette fonctionnalité permet à de nouveaux systèmes de fichiers d'être exploités sans à avoir à recompiler les applications.

La couche du système de fichiers qui réceptionne la requête va effectuer une traduction afin de déterminer les «blocs» fichiers sur lesquels porte l'accès. Le «bloc» fichier est une abstraction mise en œuvre par les systèmes de fichiers afin de manipuler les données à une granularité optimale selon le but pour lequel ils ont été conçus. La taille du bloc est comprise entre la

[14] *Virtual File System*.

taille d'un secteur disque et la taille d'une page mémoire (les plus couramment utilisées par les systèmes de fichiers étant de 512 octets, 1 Ko et 4 ko).

La totalité de la structure d'un système de fichiers est une combinaison de blocs unitaires [Bac86]. De manière simplifiée, 3 types de bloc permettent de construire une structure complète :

- Le super-bloc, c'est le bloc primordial duquel découle la totalité de la structure sous-jacente. Il contient les informations comme la taille de la partition, la taille d'un bloc, le nombre de blocs, le nombre de blocs de type *inode* libres, le nombre de blocs de type données libres, ….
- La table des *inodes*, c'est une série de blocs qui suit le super-bloc. Les *inodes* permet d'identifier de manière unique chaque fichier. Lorsqu'un usager se réfère à un fichier par un nom textuel, une correspondance est réalisée par le système de fichiers avec un identifiant numérique, c'est l'*inode*. Chaque bloc *inode* contient les méta-informations sur le fichier et des pointeurs de redirection vers les blocs de données.
- Les blocs de données, ce sont les blocs où sont stockées les données effectives de chaque fichier.

Le rôle principal d'un système de fichiers consiste à traduire des accès fondés sur des descripteurs de fichiers et des *offsets* en requêtes «blocs». Ces requêtes sont alors transmises à la couche «bloc» si elles ne peuvent pas être satisfaites par le gestionnaire de cache.

Le gestionnaire de cache

Le but d'un *cache* (ou *buffer-cache*) est de minimiser les E/S en stockant au sein de la mémoire principale les données qui devraient dans le cas contraire être servies par les disques. L'idée principale reste la même que celle evoquée lors de la présentation de la logique interne aux disques durs. Elle repose sur fait que le temps d'accès à la mémoire est beaucoup plus rapide (cf. figure 1.1) et que si une requête d'E/S intéragit uniquement avec le cache, les performances seront considérablement améliorées.

Utilisation du gestionnaire :

Lors d'un accès en lecture, si la requête «bloc» n'est pas présente dans le *buffer-cache* [15] , la requête est transmise au niveau inférieur. Lorsque la couche «bloc» retourne le ou les blocs, les données sont copiées dans une page du cache spécialement allouée. Nous rappelons que selon la granularité plusieurs blocs peuvent être contenus dans une page mémoire.

En ce qui concerne les écritures, une stratégie à base d'écriture retardée (*Write delayed*) est généralement mise en œuvre. L'idée consiste à éviter tout interaction stérile avec une unité distante (cas des fichiers temporaires qui n'ont pas à être archivés, modifications intempestives d'un fichier par un même écrivain, …). Les «blocs» sont déposés dans le *buffer-cache* et seront propagés par la suite. Un service noyau permet d'exécuter généralement cette tâche en arrière plan.

Dans le système *GNU/Linux*, le démon `pdflush` a la charge de cette opération. Il est activé lorsque la quantité de données à propager vers le disque est suffisament importante ou que l'espace libre dans le *buffer-cache* [16] est insuffisant.

[15] Le terme de *cache miss* (ou défaut de cache) est alors employé au contraire du terme *cache hit* lorsque la donnée est satisfaite par le cache.

[16] le *page-cache* depuis les noyaux *Linux* 2.4.

Techniques de pré-chargement :

Même si l'espace mémoire disponible est plus conséquent que celui présent sur le *cache* interne d'un disque, les problèmes à résoudre restent les mêmes : déterminer quels sont les blocs qui doivent rester en mémoire, quels sont ceux qui peuvent être supprimés et quels sont ceux à pré-charger.

Plusieurs algorithmes ont été proposés afin de répondre aux deux premières questions [BGH05]. Cependant l'algorithme *Least Recently Used* [CH81] et ses variantes sont généralement les solutions mises en œuvre. Lorsque le *buffer-cache* est saturé, la politique *LRU* retire la portion de données la moins récemment utilisée. C'est la solution qui est mise en œuvre au sein du noyau *Linux*.

Le pré-chargement (*read-ahead* ou *prefetch*) est un mécanisme qui consiste à réaliser par anticipation certaines requêtes d'E/S de manière à ce que les données soient présentes au sein du cache lorsque l'application en aura réellement l'utilité.

Le premier moyen de mettre en œuvre ce concept consiste à exploiter des routines asynchrones fournies par les systèmes de fichiers. L'application transmet une demande ; pendant le temps de pré-chargement, elle réalise une phase de calcul. Au moment où elle a besoin des données, celles ci sont présentes dans le gestionnaire de cache. Ces approches performantes nécessitent cependant que le développeur découpe son application selon les différentes phases de préchargement ce qui parfois peut se révéler délicat à réaliser.

La seconde solution consiste à tenter de détecter durant l'exécution d'un programme, les futures requêtes. Pour cela, le système s'appuie sur les modes d'accès utilisés par les applications. De manière simplifiée, le service en charge du pré-chargement déclenche des opérations par anticipation s'il détecte que le fichier est lu séquentiellement. En s'appuyant sur les derniers accès, il maintient pour chaque fichier ouvert un système à deux fenêtres glissantes : la fenêtre en cours et la fenêtre de préchargement. Si les accès futurs appartiennent à la fenêtre de préchargement, la stratégie *read-ahead* est maintenue. Dans l'autre cas, elle est désactivée.

Par ailleurs, le mécanisme de pré-chargement exploite le concept de *clustering* lorsqu'il doit rappatrier un bloc depuis le disque vers le gestionnaire de cache. La récupération d'un bloc ou d'une suite de blocs contigus ayant un coût équivalent, le système va tenter de récupérer l'ensemble des blocs adjacents au bloc qui doit être rappatrié s'ils correspondent au même fichier. Cette technique permet de fournir le mécanisme de pré-chargement à moindre coût.

Néanmoins, l'usage de cette approche ne se révèle pas toujours bénéfique : le chargement de blocs impliquant le «déchargement» (la suppression) d'autres. La mise en place de techniques de remplacement judicieuces devient beaucoup plus complexe [BGH05].

La couche «bloc»

Cette couche [17] a pour principal but de faire correspondre la plus petite taille adressable sur un disque, à savoir un secteur, avec la plus petite taille addressable par un système de fichier, à savoir un bloc. A la manière de la *VFS*, cette couche permet d'abstraire les différents pilotes nécessaires pour l'interaction avec les disques.

Pour chaque périphérique dit «bloc», la couche «bloc» maintient une queue des requêtes en attentes d'E/S. Les requêtes sont insérées dans les différentes queues par les couches supérieures du noyau (la couche «système de fichiers», le gestionnaire de cache ou parfois directement depuis l'espace utilisateur en utilisant des routines spécifiques). Pour chaque queue

[17] Connu sous le nom de *I/O block layer* dans les systèmes *GNU/Linux*

non vide, le pilote associé au périphérique traite les requêtes une à une. De manière similaire une fois encore aux disques, une stratégie *FIFO* ne permet pas toujours de profiter au maximum des capacités pouvant être délivrées par un disque. Ainsi, plusieurs algorithmes d'ordonnancement ont été proposés afin d'optimiser généralement le critère de performance. Nous abordons cet aspect au sein du paragraphe suivant.

Les ordonnanceurs bas niveau :
Les stratégies d'ordonnancement mises en œuvre à ce niveau consistent dans la majeure partie des cas à minimiser les déplacements des têtes de lectures/écritures de l'unité de stockage. Pour chacune des queues associées à un périphérique, un ordonnanceur sélectionne selon sa stratégie la meilleure requête. Plusieurs algorithmes ont été proposés par la communauté (*FIFO*, *SCAN*, ...) [SCO90, Tan01].

Puisque notre étude est principalement basée sur les systèmes *GNU/Linux*, nous allons présenter les quatre algorithmes d'ordonnancement disponibles au sein de ce noyau [Lov05] à savoir : *deadline*, *anticipatory*, *cfq* et *noop*. De plus, les quatres stratégies ont été évaluées lors de comportement d'accès parallèles (cf. section 4.3.2), il nous semble donc important d'en expliquer les fonctionnements.

Le premier algorithme qui a été proposé au sein des noyaux *Linux* consistait à maintenir une liste triée selon l'emplacement physique du bloc sur le disque (*Linus elevator*). Cette solution qui engendrait des famines a été remplacée par une stratégie permettant d'associer à chacune des requêtes une échéance de traitement. Cet algorithme, *deadline*, utilise deux listes supplémentaires (une pour les accès en lecture et une pour les accès en écriture) triées de manière chronologique. L'ordonnanceur vérifie à la tête de chaque liste, si la requête a atteint le délai critique associé à la queue. Si c'est le cas, la requête est immédiatement traitée. Dans l'autre cas, la requête en tête de la liste ordonnée selon l'emplacement physique est transmise au périphérique. Les délais critiques pour la liste des lectures et celle des écritures sont définis respectivement à 500 msec et 5 sec.

L'algorithme *deadline* a été amélioré afin de réduire encore le nombre de déplacements [ID01]. L'optimisation apportée consiste à attendre un bref délai avant de choisir la prochaine requête. Chaque fois qu'une application finit d'exécuter une requête, l'ordonnanceur attend un bref délai avant de recommencer son processus de sélection. Ce délai va permettre d'augmenter les chances de recevoir les requêtes suivantes de la même application, ces requêtes étant localisées potentiellement sur des blocs adjacents au niveau du disque. Un ordonnancement de ce type est appelé «oisif», nous présentons ce concept emprunté de l'ordonnancement théorique en section 5.2.2. D'une manière générale, le temps d'attente est relativement faible (6 ms pour *Linux*), il est calculé sur le coût d'un éventuel déplacement et plusieurs statistiques sur les précédents résultats de l'anticipation. Cette stratégie, intitulée *anticipatory scheduling* est la politique par défaut associée aux périphériques de type disque dans le noyau *Linux*.

L'avant dernier algorithme se concentre non plus sur le critère d'efficacité mais sur un critère d'équité. L'algorithme *Complete Fair Queuing* associe à chaque processus une liste ordonnée selon le placement physique des blocs. L'ordonnanceur sélectionne de manière circulaire une requête de chacune des listes (*Round Robin*).

Le dernier algorithme, intitulé *noop* [18] n'effectue aucun réordonnancement. La seule action qu'il effectue consiste à agréger les requêtes qui sont physiquement adjacentes au sein de la queue. Ainsi, l'ordre de sélection des requêtes correspond grossièrement à une approche *FIFO*.

[18] *No Operation*.

Cet algorithme est relativement adapté aux périphériques comme les mémoires «flash», qui ne sont pas réellement assujetties au problème de repositionnement.

Abstraction de périphériques :
La couche «bloc» propose plusieurs composants complémentaires permettant d'abstraire des périphériques. C'est parmi ces sous modules que nous allons pouvoir trouver le *RAID* logiciel, le gestionnaire de volumes logiques ou encore les «disques réseaux» (*Network Block Device*). Ce dernier composant permet d'accèder de manière transparente à une unité de stockage distante. Les messages échangés sur le réseau ont la granularité du bloc disque. Plusieurs travaux visant à optimiser les performances lors d'accès aux données en milieu distribué ont été réalisés à ce niveau. Le système «read2» [CRU03] exploite par exemple des caractéristiques des cartes Myrinet afin d'accéder directement aux contrôleurs disques distants et améliorer, ainsi, considérablement les performances.

Un modèle à base d'objet :
Située au même niveau que la couche «bloc», une couche «objet» a été proposée. L'idée consiste à déporter la gestion des blocs (allocation, libération, placement, ...) au niveau des disques. La notion d'*OBSD* [19] est alors employée [MGR03]. Les premières approches ont été proposées au début des années 1980 [RS]. Basé sur le modèle de programmation, un objet est identifié par des caractéristiques (les attributs) et propose un ensemble de fonctionnalités (les méthodes). C'est au disque «intelligent» d'exploiter ces informations afin de gérer les interactions et les évolutions de l'objet (changement de taille, extraction d'informations, ...). Le système d'exploitation est ainsi soulagé de ces démarches. Cependant, à notre connaissance et pour le moment, aucune unité de stockage n'intègre ce modèle en natif. Nous avons décrit rapidement cette couche car le système de fichiers *Lustre*, présenté par la suite, s'appuie sur ce modèle (cf. section 3.1.8).

1.3. Bilan

Les évolutions technologiques ont permis de fournir des disques proposant de très larges capacités de stockage. Couplée aux grandes puissances d'analyse disponibles dans les centres de calculs, les nouvelles applications exploitent cet espace et manipulent une quantité de données de plus en plus importante. Cependant, les temps d'accès proposés par les unités de stockages actuelles restent très élevés. Cette opération est la plus coûteuse au sein d'une architecture informatique aujourd'hui (9 ms environ).

Ainsi, plusieurs techniques logicielles sont mises en œuvre afin de pallier cet écart de performance. Basé sur une superposition de couches, la pile des E/S tente d'apporter à différents niveaux plusieurs solutions logicielles permettant de réduire les interactions avec les disques physiques ou d'optimiser les accès lorsque ces derniers sont obligatoires.

Le gestionnaire de cache qui d'un point de vue conceptuel peut être retiré propose, par exemple, un grand nombre de mécanismes permettant de réduire les latences liées aux E/S.

L'utilisation d'algorithmes de pré-chargement présents tant au niveau des disques qu'au niveau de la pile des E/S sont employés à ce but. Cependant, ils reposent sur la détection des

[19] *Object-Based Storage Devices*.

accès séquentiels au sein d'une application. Or, comme nous allons le voir dans le chapitre suivant, les modes d'accès mis en œuvre par les nouvelles applications parallèles ne correspondent pas toujours à ce modèle. Au contraire, le comportement parallèle propre à ces programmes a un impact significatif sur les performances des disques en générant potentiellement un grand nombre de déplacements.

LES ENTRÉES/SORTIES ET LE CALCUL INTENSIF 2

L'informatique a fourni le moyen de traiter rapidement et de conserver des informations dans tous les domaines d'activités. La quantité de données générée par certains développements modernes (climatologie, biologie moléculaire, génomique, physique des hautes énergies, traitements d'images ...) s'avère gigantesque. Or, comme nous l'avons vu au chapitre précédent, la manipulation de données reste une opération coûteuse et de nombreux travaux ont révélé que les performances de telles applications parallèles sont intimement liées aux méthodes et aux politiques de gestion de données choisies.

Afin de comprendre cette forte dépendance vis à vis des systèmes de stockage, nous allons décrire, dans ce chapitre, le comportement général et les modes d'accès des E/S au sein des applications *HPC* .

Cette étape va permettre au lecteur de mieux appréhender l'intérêt des différentes techniques mise en œuvre afin de réduire l'impact sur les performances. Ces mécanismes sont abordés dans une seconde partie.

2.1. Caractérisation des E/S

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 [CACR95, NKP+96, May01] et a permis de définir de manière non exhaustive l'ensemble des besoins E/S.

2.1.1. Généralités

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

Avant d'entrer plus amplement dans le sujet, nous allons définir plusieurs termes qui permettront de faciliter la compréhension du document par la suite :

Fichier :
: 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.

Descripteur de fichier :
: Quand un processus d'une application ouvre un fichier, le système d'exploitation lui retourne un identifiant, 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 processus sont exécutées en utilisant ce descripteur de fichier. Chaque fois qu'une application lit ou écrit (`read/write`), le système de fichiers repositionne le pointeur 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.

Application parallèle :
: Une application multi-tâches (*threads* ou processus *SPMD*[1]) qui travaille en concurrence afin de résoudre un problème. Nous ferons la distinction entre les applications parallèles peu dépendantes du système de stockage (l'application est alors qualifiée de *CPU-bounded*) et celles fortement dépendantes des E/S (applications *I/O-bounded* [2]).

Architecture parallèle distribuée :
: Support physique permettant d'éxécuter une ou plusieurs applications parallèles. Nous différencierons les nœuds multi-processeurs des architectures de types grappes visant à interconnecter des machines mono ou multi-processeurs.
Dans un tel contexte, une application parallèle peut être exécutée selon ses besoins sur une ou plusieurs machines. Cette distribution à une échelle supplémentaire, complexifie la mise en œuvre de solutions performantes pour les E/S parallèles (et ce même si la problématique reste identique). Par exemple, même si chacune des machines possède son propre espace de stockage, l'utilisation d'un système de fichiers global est généralement mise en œuvre. Un tel système permet à une application d'accéder et de partager des fichiers de manière répartie. Une grappe peut être alors vue comme l'interconnexion de deux groupes : les nœuds dit de «calcul» où sont exécutées les applications et les nœuds dit de «stockage» exploités pour la sauvegarde (cf. figure 2.1).

[1] *Single Program Multiple Data*, une tâche `MPI` par exemple.
[2] *Input/Output*.

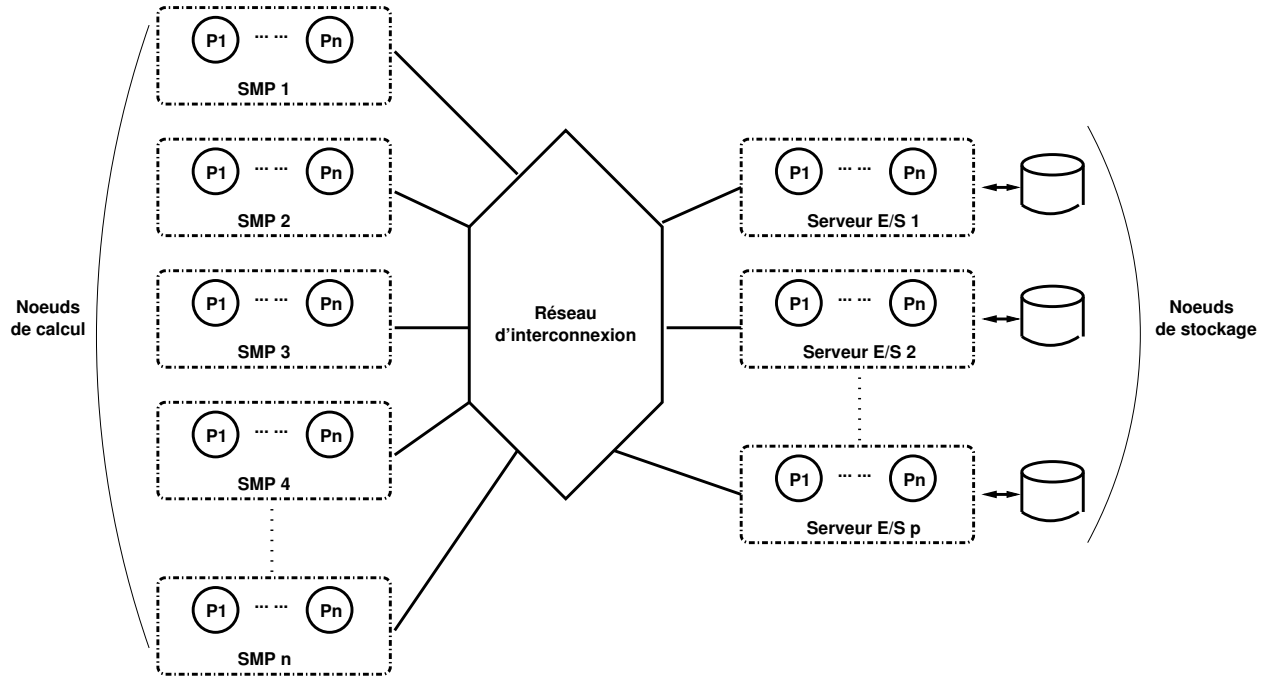


FIG. 2.1 – Représentation d'une architecture de type grappe
Cette vision simplifiée va nous permettre d'indiquer par la suite où sont situés les points critiques et où sont réalisés les différents mécanismes mises en œuvre pour y faire face.

E/S parallèles :
: Accès aux données (lecture et/ou écriture) par une application de manière parallèle (ou simultanée). Cette notion de parallélisme peut être amplifiée selon l'architecture matérielle employée pour le stockage de données (plusieurs disques, plusieurs serveurs ...). Chacun des processus de l'application accède au même fichier sur une même période et à différents endroits. Du fait qu'il posséde généralement chacun leur propre descripteur de fichier, il est primordiale de définir la sémantique utilisée pour assurer la cohérence [Leb02] lors d'accès de type écriture. Les systèmes de fichiers locaux implémentent généralement une cohérence forte : toute modification qui survient sur un bloc (cf. section 1.2.2.3) 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.

Après avoir rappeler plusieurs définitions fondamentales, nous allons nous attarder de manière plus spécifique sur les besoins des applications au niveau des E/S disques. Si nous avons vu, dans le chapitre précédent, qu'un système de fichiers peut être fonctionnellement divisé en plusieurs parties, nous allons voir que la "couche haute" qui fournit un ensemble de services aux clients (primitive d'ouverture, de création et d'accès aux ressources) est souvent incomplète et inadaptée à la programmation parallèle.

2.1.2. Écriture, lecture et taille

Les trois principales méthodes d'ouverture d'un fichier sont :
- en lecture, les opérations réalisées seront uniquement de type lecture ;
- en écriture, les opérations réalisées seront uniquement de type écriture ;
- en lecture/écriture, les opérations réalisées seront de type lecture ou écriture.

Nous insisterons sur la notion de fichier temporaire [3] : tout fichier qui a été créé et détruit par un même nœud durant la même tâche.

D'une manière générale, il a été montré que la quasi-totalité des fichiers sur des systèmes usuels (PC familial, station de travail ...) ont une taille inférieure à 1 Mo. A contrario, les applications scientifiques fortement dépendantes des E/S génèrent des fichiers de taille bien supérieure. Par ailleurs, les fichiers de type «lecture» sont de dimension conséquente alors que les fichiers de type «écriture» sont pour la majeure partie, de taille inférieure.

Par ailleurs, l'ensemble des accès sur un même fichier est souvent identique (très peu de ressources sont ouvertes afin de réaliser des opérations à la fois d'écriture et de lecture et il y a très rarement plus de quatre fichiers ouverts à la fois). Les opérations de lecture représentent plus de la majorité des accès réalisés dans le cadre des applications parallèles.

D'un point de vue global, il y a plus de fichiers de type «écriture» que de type «lecture». Cela s'explique par la forte utilisation de fichiers temporaires sur chaque nœud (à défaut de centraliser les données au sein d'un même fichier). En effet, cette technique permet d'augmenter les performances d'E/S en utilisant les périphériques de stockages locaux et en évitant les problèmes d'accès concurrents. Cependant, la programmation parallèle a imposé un plus haut degré de partage au sein d'une même ressource et il est parfois impératif de centraliser l'information en un point unique. Les problématiques d'accès concurrents souvent évitées au sein des systèmes de fichiers «utilisateurs» [4] doivent être prises en compte dans le cadre d'applications fortement dépendantes des E/S. Ainsi, un nouveau mode d'ouverture de fichier dit «partagé» a été identifié, il permet à un ensemble de processus d'exploiter le même descripteur de fichier. Après avoir effectué de manière commune l'ouverture de la ressource, toute opération réalisée par la suite aura un impact sur l'ensemble des processus partageant le fichier.

2.1.3. Classification selon la dépendance aux E/S

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

- Compulsive :
  L'ensemble des accès est réalisé simultanément : 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 forts débits et un équilibrage optimal des accès pour limiter au

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

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

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

- Contrôle :
  les accès d'E/S 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 contrôle (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'E/S 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 périphériques de stockage secondaire sont dûs à la limitation de la taille de la mémoire principale. L'utilisation du concept de mémoire virtuelle [5] a permis de résoudre cette restriction. Néanmoins, cette approche à un coût non négligeable et se révèle être trop lente pour plusieurs codes de calcul. De nombreux problèmes scientifiques ont des quantités de données trop grandes pour être contenues dans l'espace de mémoire principal. Ainsi, l'augmentation de la mémoire vive, qui peut être parfois une solution temporaire, devient très vite onéreuse en plus d'avoir une limitation technique liée au nombre de bancs *RAM*.

2.1.4. Mode d'accès

Débuté en 1993, le projet CHARISMA [6] [NKP+96], a consisté à identifier les différents types d'accès utilisés par les applications *HPC*. Plusieurs expérimentations ont permis de déterminer des comportement d'accès récurrents. Les premiers modes sont habituellement connus et ne sont pas uniquement propres aux applications scientifiques.
Il s'agit des modes :

- séquentiel :
  Un comportement est considéré séquentiel si chaque nouvelle requête débute à un *offset* supérieur au dernier appel.
- contigu :
  Un comportement est dit contigu (ou consécutif) si chaque accès débute exactement à l'*offset* sur lequel le dernier appel s'est fini.
- aléatoire :
  Un comportement est défini comme aléatoire si aucune structure régulière ne peut être établie. Ce genre de mode est généralement celui qui correspond aux requêtes générées par un système de gestion de base de données. Chacune des requêtes est dépendante des critères de sélection et l'ensemble des accès est dispersé sur la totalité du fichier.

[5] Une partie de l'espace de stockage secondaire est utilisée afin d'augmenter la taille de la mémoire primaire.
[6] *CHARacterize I/O in Scientific Multiprocessor Applications*.

A la différence des modes d'accès des bases de données, il a été montré une certaine régularité dans les accès émanant des applications scientifiques parallèles :

- Régularité des intervalles entre chaque requête :
  La taille de l'intervalle entre chaque requête semble être régulière.
- Régularité dans la taille de données demandée à chaque requête :
  La majorité des accès parallèles au sein d'un fichier se réalise sur la même quantité de données.

En effet, les applications *HPC* partitionnent leurs accès en se calquant dans la majeure partie des cas sur le découpage parallèle physique de l'application [7] . En combinant ces deux phénomènes, un grand nombre d'applications utilise un modèle [8] d'accès aux données clairement structuré.

Un *stride* est un groupe de requêtes constituant un modèle d'accès régulier. Deux niveaux ont été définis :

- Accès disjoint régulier (ou simple) :
  Tous les accès s'effectuent sur une quantité de données identique et chaque accès est séparé par le même espace (le modèle est constitué d'un unique *stride*).
- Accès disjoint de niveau n (double, triple ...) :
  Le modèle d'accès [9] est composé de plusieurs *stride* séparés par un même espace.

La répartition d'une matrice tri-dimensionnelle sur p processeurs illustre parfaitement ce genre d'accès. Lorsque la distribution des colonnes est équitable sur l'ensemble des nœuds, le modèle d'accès est de type *simple*. Dans le cas inverse, le modèle est plus complexe et se compose de plusieurs segments de type *stride*, il est *double*.

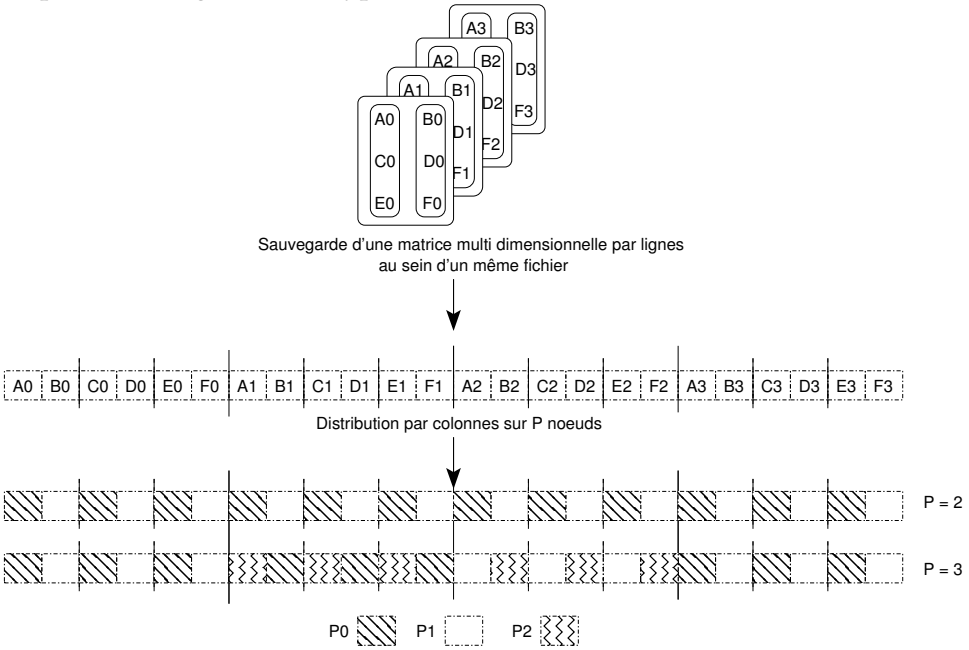


FIG. 2.2 – Distribution par colonnes d'une matrice sur un ensemble P de processeurs

[7] La répartition des différents processus sur les différents processeurs

[8] Communément appel *strided access pattern*.

[9] *Nested pattern*.

Dans la figure 2.2, pour $p = 2$, le modèle d'accès est constitué d'un unique segment qui se décompose, après un déplacement initial [10] de la même manière pour P0 et P1 :

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

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

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

2.1.5. Disparité entre données en mémoire et fichiers disques

Un aspect important lors de la manipulation performante de données par une application consiste à prendre en compte la différence entre la distribution des données en mémoire et leurs placements physiques sur le disque [NL01].

Il est en effet impératif de considérer, lors du processus de partage de données, deux niveaux (cf. figure 2.3) :

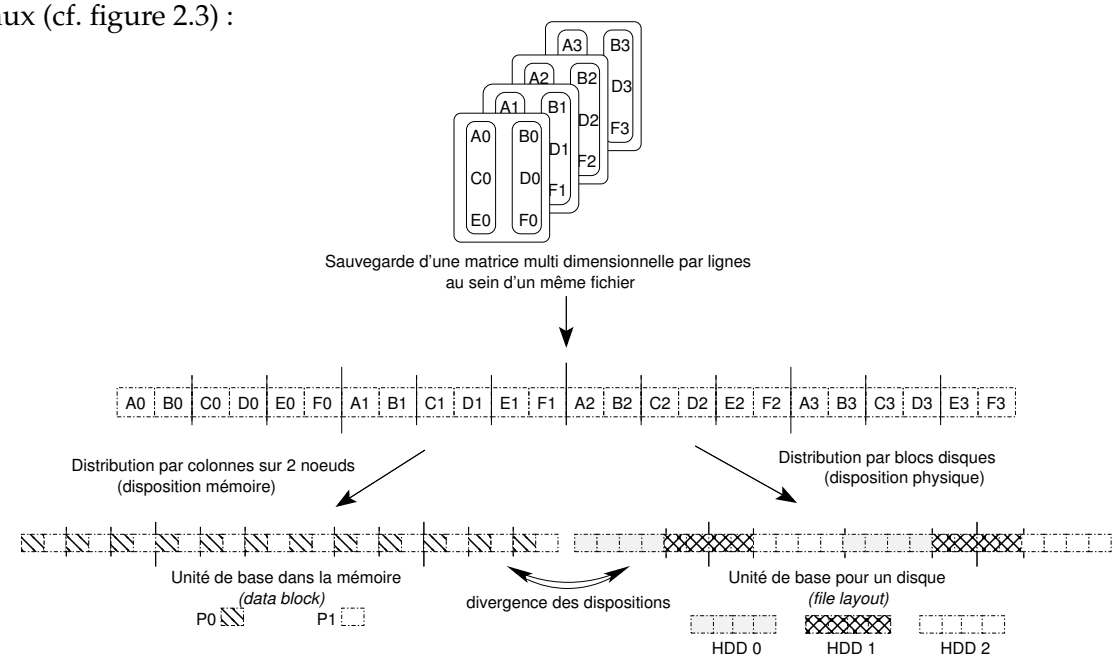


FIG. 2.3 – Vue logique (fichier) *vs* Vue physique (blocs)
Différence au niveau du placement des données (contiguïté mémoire *vs* contiguïté physique / unité d'accès mémoire *vs* disque).

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

[10] Nous remarquerons que les masques sont la plupart du temps équivalent pour l'ensemble des nœuds, seul diffère l'*offset* de départ sur lequel est initialisé le pointeur de fichier. Sur la figure 2.2, P0 et P1 utilisent le même masque ; toutefois le modèle pour P0 débute à l'*offset* 0 alors que celui de P1 est décalé d'un bloc.

– 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. L'accès aux données peut alors être fait 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 un rôle important pour les performances globale du système.

2.1.6. Bilan

La grande difficulté dans la gestion des E/S parallèles est que l'ensemble des requêtes est transmis au système de fichiers de manière simultanée et ce depuis un nombre potentiellement conséquent de processus s'exécutant sur un ou plusieurs nœuds. Ce mode d'accès engendre un surcoût lié au grand nombre de demandes et plusieurs phénomènes de congestion peuvent apparaître :

– au niveau des clients, lorsque plusieurs processus accèdent à la pile des E/S,
– au niveau des serveurs, lorsqu'une quantité importante de requêtes doit être traitée,
– sur les unités de stockages qui repositionnent sans cesse les têtes de lecture/écriture.
– au niveau du réseau d'interconnexion qui doit être capable de soutenir la charge imposée par les nombreux échanges de messages.

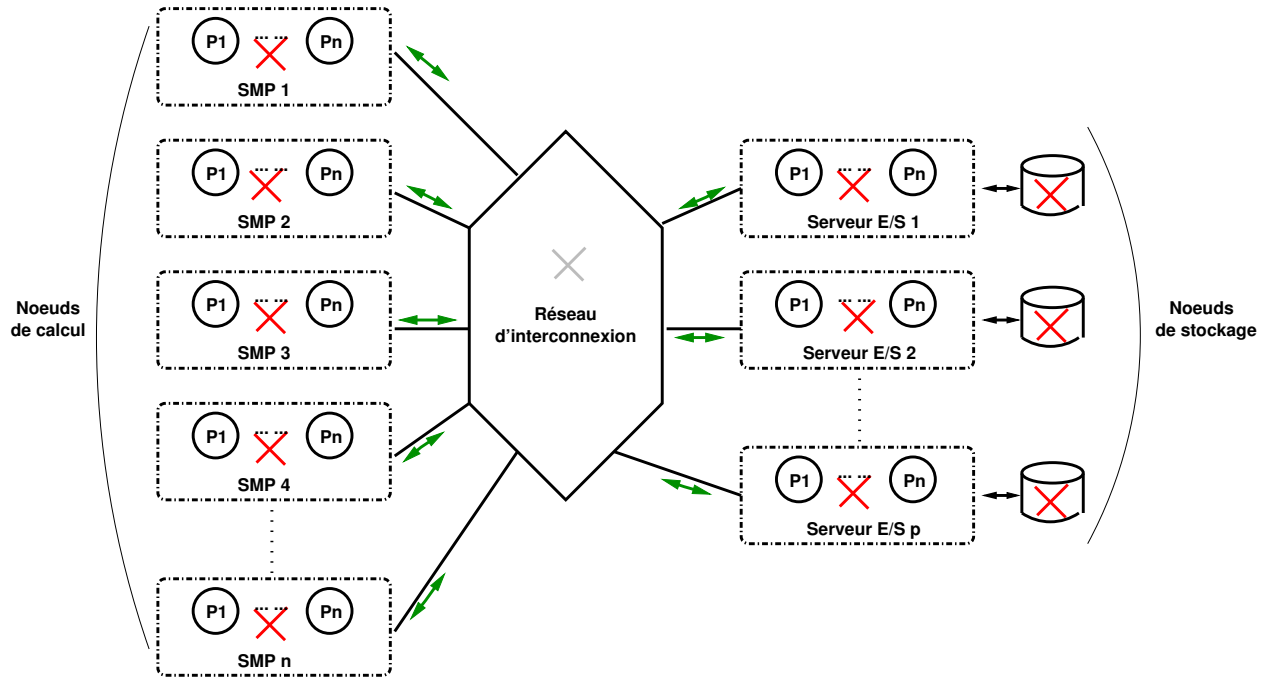


FIG. 2.4 – Points susceptibles d'avoir un impact sur les performances des systèmes d' E/S

Par ailleurs, nous rappelons que les solutions de pré-chargement sont basées sur l'arrivée séquentielle des requêtes. Ainsi, même si une structure régulière entre les accès existe, l'envoi des messages et le décalage entre les différents processus complexifient le réordonnancement des requêtes par une couche système coté serveur de stockage et les mécanismes de

pré-chargement propres à la pile d'E/S se révèlent être la plupart du temps inadaptés. Depuis les couches basses (côte serveur, cf. couche 1.2.2.3), le mode d'accès correspond alors à un comportement aléatoire et il est impossible d'y apporter des optimisations. Un nombre important de requêtes disjointes doit être servi par l'unité de stockage qui, par conséquent, repositionne les têtes de lecture/écriture à chaque étape. Les performances sont ainsi dégradées.

En s'appuyant sur les structures régulières [11] , présentes au sein des applications parallélisées, plusieurs mécanismes ont été proposés afin de reconstruire un mode d'accès efficace. Nous abordons les plus connues dans la prochaine section.

2.2. Optimisation des comportements parallèles

Les unités de stockage de type disque offrent des performances optimales lors d'accès conséquents : opérations de lecture (ou d'écriture) réalisées sur une grande quantité de données en un unique accès (9 ms pour se positionner et seulement 3 ms pour transférer un bloc de 64Ko [ID01]). Or, nous avons indiqué que les modèles d'E/S au sein des programmes parallèles *HPC* consistent la plupart du temps en un grand nombre de requêtes simultanées sur des quantités plus ou moins grandes et de plus non contigües.

Une approche, certes naïve mais qui reste largement employée, consiste à faire pré-charger par un processus l'ensemble des données utiles à l'application dans son espace mémoire. Les données seront alors redistribuées au moment opportun vers les processus concernés. En plus d'un réseau d'interconnexion rapide (les données transitant par deux fois sur le réseau), cette solution dite de pré-chargement nécessite une taille d'espace mémoire assez conséquente pour supporter le fichier. Dans le cas contraire, l'application se confronte à un problème *out-of-core* dépendant également des E/S.

Afin de pallier à ce problème d'E/S parallèles, quatre approches ont été proposées par la communauté scientifique : l'agrégation des accès, l'utilisation de techniques asynchrones, la mise en place de techniques de caches et/ou d'ordonnancement. Comme nous allons le voir, chacun des mécanismes peut être mis en œuvre à différents endroits (niveau applicatif *vs* couches systèmes ou encore côté clients *vs* serveurs). Les possibilités d'optimisation étant dépendantes de la localisation, chacune des techniques doit réaliser des compromis. Nous allons présenter les techniques les plus utilisées dans la suite de ce chapitre.

2.2.1. Agrégation des accès

Cette technique consiste à agréger plusieurs requêtes en une unique plus conséquente. Le but est double : il permet, en effet, de diminuer le nombre de requêtes à destination du système de fichiers et de réaliser des accès plus conséquents et donc plus performants.

La majeure partie des solutions disponibles pour améliorer les performances repose sur cette idée [TGL02, TGL99a, NL97, LR96, MWLY03, HJMY00]. Nous allons présenter au sein de cette section les différences majeures entre chacune d'entre elles. Dans un premier temps, nous décrirons deux mécanismes exploités au sein d'un processus (et donc d'un unique nœud de calcul) puis nous nous attarderons sur des stratégies collectives permettant d'agréger l'ensemble des requêtes d'une application parallélisée.

[11] Nous employerons par la suite, le terme de «schéma d'accès» ou «vue» (par analogie aux vues *SQL*).

Vecteurs d'E/S

La première solution pour transmettre plusieurs requêtes en un appel consiste à utiliser les routines à vecteur d'accès. Ce genre d'approche est réalisé au niveau applicatif sur les nœuds clients.

L'interface `POSIX` fournit les fonctions `readv()` et `writev()` qui permettent en un appel de manipuler plusieurs zones mémoires disjointes situés en espace utilisateur depuis/vers une portion contigüe d'un fichier. Ce premier mécanisme ne permet malheureusement pas d'effectuer la tâche inverse : manipuler une zone contigüe depuis/vers de multiples zones au sein d'un fichier.

L'approche *List I/O* [CCC+03] a été proposée afin d'y remédier. Elle permet d'indiquer au sein d'un unique appel un schéma d'accès complexe : simple, double, ... 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 2.5 présente cette technique.

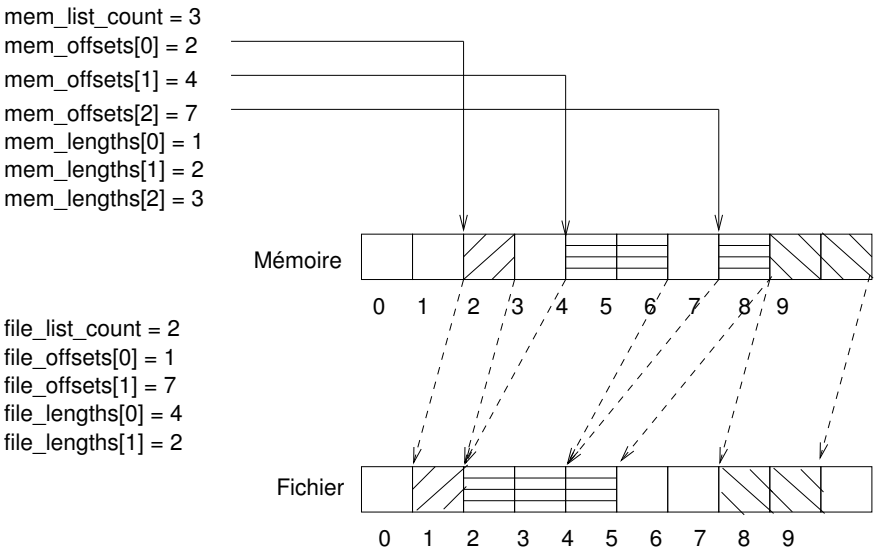


FIG. 2.5 – Exemple d'écriture utilisant l'approche"*List I/O*"
4 requêtes sont encapsulées au sein d'un unique appel.

Le principal inconvénient de cette approche est que les couches clientes du système de fichiers sous jacent doivent proposer des fonctionnalités équivalentes. Dans le cas contraire, l'accès va être de nouveau redécoupé en un accès par zone mémoire/zone disque.

Stream based I/O

Cette approche a été proposée afin de réduire les interactions qui transitent par les réseaux dans le cadre de systèmes de fichiers distribués. Développé dans le cadre du projet *PVFS* ([LR96], cf. section 3.1.6), le concept consiste à agréger au niveau des nœuds clients plusieurs requêtes au sein d'une même trame réseau afin de l'envoyer par la suite au serveur de données. Le nombre de messages transitant ainsi sur la grappe est considérablement diminué.

La construction de chaque trame "de requêtes" prend en compte le protocole d'échange employé et la taille optimale des segments correspondants. Ainsi, cette méthode permet de réduire considérablement les messages de contrôle nécessaires à chaque requête d'E/S et donc le transit réseau.

L'opération d'empaquetage des accès réalisée au sein des couches systèmes côté client (couche propre au système *PVFS*) et la phase de restructuration des requêtes (côté serveur) représentent les surcoûts de cette méthode.

Néanmoins, cette solution ne prend pas en compte les problématiques liées au placement et à l'entrelacement des données (il est possible d'effectuer plusieurs requêtes sur des mêmes données). De plus, elle a été proposée en s'appuyant sur une architecture où la latence réseau est relativement élevée par rapport aux performances délivrées des unités de stockage ce qui, à ce jour se révèle, comme nous l'avons abordé au chapitre précédent, souvent inexact.

Data Sieving

Le *Data Sieving* est une technique permettant d'agréger plusieurs requêtes non contiguës provenant d'un même processus. A la différence des vecteurs d'accès ou de l'approche *Stream based I/O* où seules les données utiles sont manipulées, l'approche *Data Sieving* consiste à réaliser un seul appel qui recouvre l'ensemble des sous requêtes.

L'exemple ci-après illustre le principe de cette méthode : supposons que nous souhaitions réaliser 5 requêtes de lecture à partir d'un même processus et que ces accès ne soient pas contigus. Au lieu de transmettre 5 appels, un unique appel portant depuis la position de début jusqu'à la de fin de ces requêtes est calculé et une unique opération de lecture est réalisée. La totalité des données est placée dans un tampon temporaire avant d'être redistribuée vers les diverses zones prévues par l'application.

La partie gauche de la figure 2.6 illustre ces propos.

Cette solution est limitée par deux principaux facteurs :

- La taille du tampon de recopie temporaire :
  Elle définit le nombre d'appels nécessaire à exécuter afin de «recouvrir» l'ensemble des blocs distants.
- Le degré de non continuité :
  Il définit l'espace entre deux blocs distants. Ainsi, si ce facteur est trop élevé, la mise en œuvre de la méthode peut s'avérer nettement plus chère que le coût cumulé de l'ensemble des accès requis pour récupérer les données utiles par de simples appels `POSIX`. Il faut bien avoir conscience qu'à la différence des deux méthodes précédentes, un nombre plus ou moins important de données «inutiles» transite entre l'application et le système de fichiers. Il a été montré qu'un calcul de ce rapport à chaque appel permet de choisir de manière optimale la stratégie à appliquer et ce sans pour autant ajouter un coût significatif [Raj02] (nous présentons de manière détaillée le calcul du degré de continuité en annexe A).

Enfin, il est intéressant de noter que si cette solution peut également être employée lors des opérations d'écritures, elle nécessite la mise en place d'un type d'accès spécifique : *read-write modified*. Ce genre de procédure permet de modifier uniquement les portions de données souhaitées (les informations comprises au sein du *stride* ne sont pas écrasées). Par ailleurs afin de garder l'intégrité des données, il est impératif de verrouiller la région du fichier à laquelle

l'application accéde. Par conséquent et afin de réduire les phénomènes de contention dûs à la prise de verrous, le tampon temporaire utilisé pour les écritures est de taille beaucoup plus faible que celui mis en œuvre lors des opérations de lecture.

Cette solution est généralement présente dans les bibliothèques applicatives mettant en œuvre les spécifications `MPI I/O` (cf. section 3.2.6).

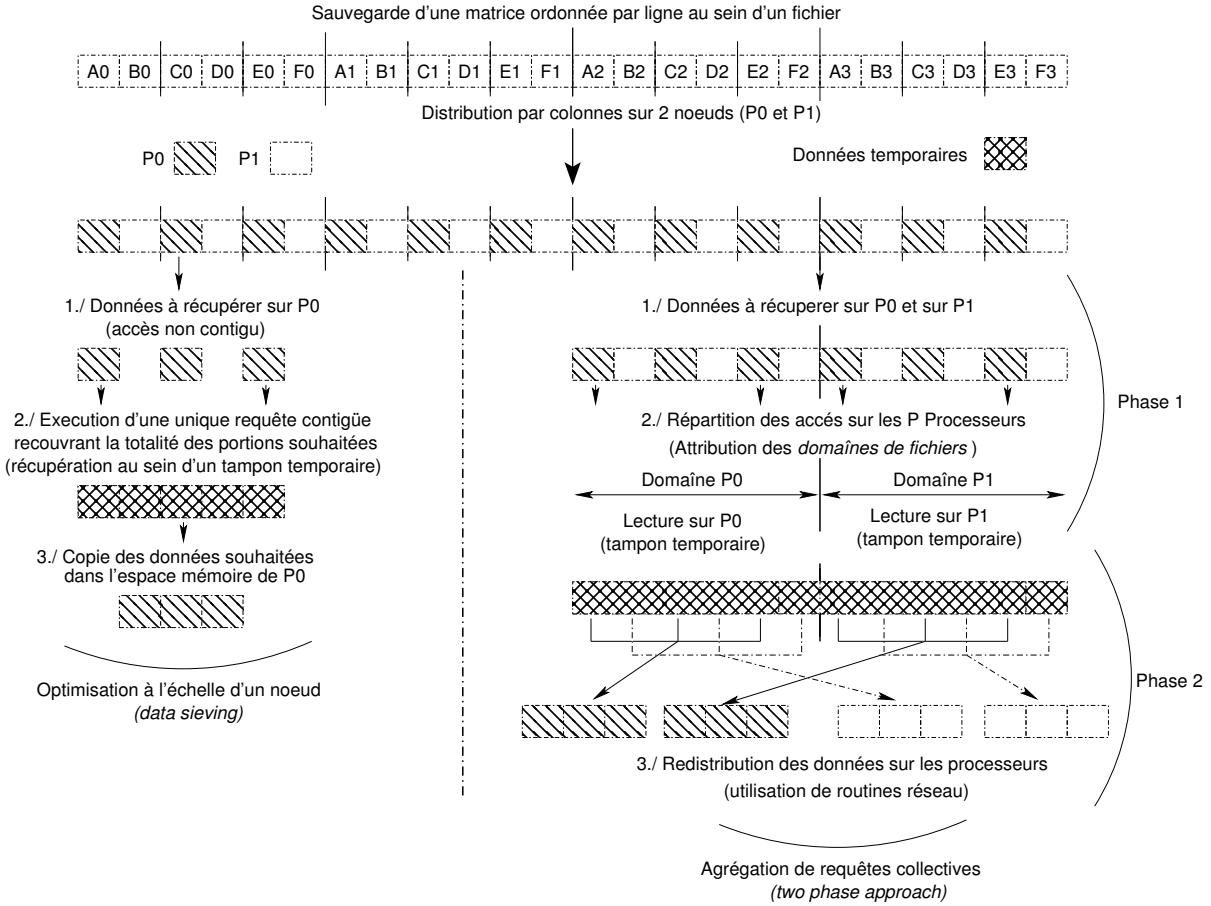


FIG. 2.6 – Optimisation des E/S lors d'accès parallèles
Optimisation dans le cadre d'une distribution par colonnes d'une matrice sur 2 processeurs.
Data Sieving à gauche et *Two Phase* à droite.

Agrégation multi-processus

A la différence du cas précédent pour lequel l'optimisation est réalisée indépendamment des autres processus, nous nous plaçons ici dans un contexte de requêtes en provenance de plusieurs processus d'une même application déployée potentiellement sur plusieurs machines. L'intérêt est de tenir compte des accès parallèles recouvrants (entrelacés) lors de demandes collectives. Une agrégation optimale de ce type de requêtes peut apporter un réel gain.

Ce concept, communément appelé *Collective I/O*, peut être mis en œuvre à plusieurs endroits, chaque solution possédant évidemment ses avantages et ses inconvénients :

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

– La première initiative est celle la plus régulièrement mise en œuvre. Elle est notamment implantée au sein de `ROMIO` (cf. section 3.2.6.1), la couche `MPI I/O` de la bibliothèque *MPICH*. Comme son nom l'indique, la méthode se divise en deux étapes successives :

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

Cette solution brièvement décrite dans la partie droite de la figure 2.6 s'appuie sur une architecture possédant un réseau d'interconnexion non pénalisant. Pour les opérations d'écriture, les deux étapes sont simplement inversées.

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

1. En tenant compte de la distribution des données en mémoire (distribution par bloc, cyclique ...) et de la taille des tampons temporaires (le but étant de minimiser l'espace mémoire requis),
2. En réduisant les flux réseau, pour cela des primitives [12] permettant d'agréger au sein d'un même message plusieurs informations (à chaque itération, un seul message contenant la totalité des blocs de données à permuter est émis vers le nœud cible au lieu d'envoyer un message pour chaque information).

D'une manière générale, il n'y a plus de réelle distinction entre ces deux méthodes. L'algorithme *Two Phase* tire partie des différentes améliorations proposées par la communauté scientifique et permet de maximiser le ratio coût/performance.

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

1. Des demandes simples [13] sont transmises sur l'ensemble des disques où elles sont agrégées afin de maximiser les performances puis enfin exécutées.
2. Les données lues sont alors directement retransmises sur le nœud souhaité.

La phase de redistribution inter-nœud ayant disparu, les performances se révèlent meilleures. Toutefois, cette approche est nettement moins portable que les méthodes soumises auparavant. Par ailleurs, elle ne diminue pas le nombre de messages réseaux. Une étude [IRU01] s'est inspirée de critères d'optimisation (localité des données, consommation restreinte des flux réseaux) afin d'améliorer l'implémentation de `ROMIO` pour le système de fichiers *PVFS*.

[12] *Scatter/Gather*.

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

Au niveau du/des serveur(s) : le modèle *server-directed I/O* a été développé dans le cadre du projet *PANDA* [14] [SCJ+95]. Cette solution correspond à une abstraction du concept *Disk Directed I/O*. En effet, si ce dernier apporte de réels gains lors des accès, il reste toutefois très dépendant de l'architecture sous-jacente. L'intérêt est de remonter ce module à un niveau applicatif améliorant ainsi la portabilité du modèle et ce, sans un surcoût considérable. Les nœuds de stockage sont généralement utilisés afin de déployer le serveur. Cependant, il est tout à fait possible d'exploiter un nœud de calcul pour supporter cette charge.

D'un point de vue général, une étude [AUB+96] a montré qu'il était possible d'obtenir les performances requises au sein des applications *HPC* sans passer par des approches de type *Collective I/O*. Il est montré que l'utilisation de techniques collectives augmente les phénomènes de latence en imposant de nombreux points de synchronisation pour débuter les différentes phases (temps pendant lequel l'unité de stockage n'est pas exploitée). S'appuyant sur quatre applications scientifiques correctement remaniées pour maximiser l'utilisation des disques («larges» requêtes, exploitation d'opérations de *Data Sieving* via la librairie *Jovian* 2, cf. section 3.2.4), il prouve qu'un gain comparable peut être apporté.

Toutefois, il est nécessaire d'intervenir au sein du code afin d'apporter les modifications permettant ces optimisations ce qui complexifie, là encore, la tâche du développeur. Il paraît donc important de prendre en compte les remarques faites concernant les opérations collectives et de continuer à travailler sur des interfaces de plus haut niveau permettant de rendre plus aisée la programmation d'applications parallèles.

2.2.2. E/S asynchrones

Les techniques d'agrégation présentées précédemment permettent d'exploiter efficacement les débits proposés par les unités de stockage actuelles. Cependant, même si les latences liées aux opérations d'E/S sont minimisées, elles restent toujours présentes.

Une approche complémentaire à l'agglomération subtile des demandes d'E/S consiste à utiliser des techniques asynchrones permettant soit de prévoir et de réaliser, à l'avance, d'éventuelles demandes ou d'exploiter diverses fonctionnalités (logiciel ou matériel) capable de réaliser des opérations en arrière plan. Nous verrons qu'un grand nombre de solutions (`Galley`, `PVFS2`, `MTIO`, `MercutIO` ou encore `ROMIO` cf. chapitre 3) met en œuvre des méthodes de ce type afin de rendre plus «transparent» le coût des opérations lié aux E/S en maximisant le recouvrement E/S *vs* Calcul.

Les fonctionnalités offrant l'asynchronisme sont intégrées de manière croissante au sein des disques de stockage. Cependant peu de systèmes exploitent d'ores-et-déjà cette capacité et les opérations asynchrones sont implémentées en s'appuyant sur des processus s'exécutant en arrière plan.

Plusieurs travaux [DT99, DT01] ont montré l'importance, là encore, d'une étude rigoureuse et d'une exploitation optimale des *threads*.

Par exemple, dans le cadre des opérations collectives fondées sur la méthode *Two-Phase*, il semble à *priori* pertinent de créer un *thread* dédié aux E/S afin de réaliser l'opération en arrière plan (le processeur continuant pendant ce temps à réaliser différents calculs). Malheureusement, cette approche s'avère souvent plus laborieuse qu'exécuter la totalité de l'opération d'E/S en premier plan.

[14] *PANDA* est une bibliothèque qui a pour but l'optimisation des E/S au sein des tableaux à plusieurs dimensions sur des architectures aussi bien séquentielles que parallèles, cf. section 3.2.2.

En effet, les étapes susceptibles d'être recouvertes dans l'algorithme *Two-Phase* correspondent à l'opération d'E/S en elle-même et aux différentes phases de redistribution des données (après l'envoi et jusqu'à la réception des messages de permutations des données). Une fois que les deux *threads* sont créés, ils rentrent en «compétition» pour le contrôle du *CPU* : le coût engendré par les changements de contexte et la forte utilisation de l'ordonnanceur n'est pas avantageux lorsque le recouvrement est réalisé sur une période trop courte.

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

D'un point de vue général, l'usage de routines asynchrones complexifie le développement et l'évolution des applications parallèles qui doivent déjà prendre en compte les problèmes de vue logique et de placement physique. Pour qu'une opération asynchrone soit pertinente, il est impératif de prévoir un code de recouvrement, ce qui peut se révèler dans certains cas difficile à mettre en place.

2.2.3. *Caches* et techniques de pré-chargement

Nous avons mentionné au chapitre précédent l'importance des gestionnaires de caches pour les performances tout en précisant la complexité des mécanismes internes qu'ils exploitent (pré-chargement, sélection des pages à supprimer, ...) au sein d'une même machine.

Dans le cadre d'une architecture parallèle distribuée, l'espace mémoire disponible pour «cacher» les données est généralement plus conséquent. Ces zones «tampon» sont mises à contribution afin de réduire l'impact sur les performances des systèmes de sauvegarde. Par exemple, l'utilisation des disques locaux est un moyen supplémentaire pour limiter les phénomènes de saturation au sein des applications fortement dépendantes des E/S : les fichiers de type écriture sont souvent sauvegardés localement et centralisés par la suite si besoin est.

Néanmoins, la mise en place d'un gestionnaire de *cache* global à l'architecture est une opération délicate qui requiert un grand nombre de mécanismes. Plusieurs concepts provenant des systèmes à mémoires partagées distribuées [Cec01] sont communément exploitées afin de proposer des solutions. Les concepts de migration ou encore de duplication sont souvent utilisés afin d'améliorer la localité des données :

- Pour la migration, il s'agit de rapatrier une copie des données qui a été chargée dans un cache distant de l'architecture vers le cache du nœud où est exécuté le processus souhaitent accéder à l'information. Cette technique, qui semble efficace dans un premier temps, peut se révéler coûteuse lorsque les données cachées ne cessent d'être migrées entre plusieurs nœuds (effet *ping-pong*).
- La duplication est une technique complémentaire utilisée lors d'accès de type lecture. Chacune des machines ayant des processus qui souhaitant accéder aux données, rapatrie une copie au sein de son gestionnaire de cache local. Cette technique permet de réduire considérablement les transferts lorsque ces derniers sont redondants, chacun des accès pouvant être satisfait par le *cache* local. Les systèmes de stockages et le réseau sont ainsi moins sollicités.

Ce genre d'approche est généralement employé pour les données initiales des programmes *HPC* compulsifs. Cependant, en plus de nécessiter d'un espace temporaire de stockage qui peut devenir rapidement important, cette technique n'a pas été conçue pour prendre en compte les accès de type écriture ou écriture/lecture. Dans de tels cas, des problèmes de cohérence apparaissent.

Un système de cache distribué requiert, d'une part, la mise à disposition d'un service permettant de localiser à tout moment les différentes copies des données et, d'autre part, de détecter les accès concurrents afin d'assurer la cohérence. Un des modèles le plus simple consiste en un protocole à écrivain unique et lecteurs multiples. Chaque accès en écriture sur une copie va invalider l'ensemble des copies qui y sont rattachées. La nouvelle copie devra donc être rapatriée à nouveau afin de prendre en compte les modifications. Maintenir ce type de cohérence est relativement couteux. Des modèles de cohérence relachée ont été proposés afin de minimiser les échanges de messages. Ces derniers requierent de la part des programmeurs de prévoir des routines de synchronisation permettant d'ordonner les accès.

La thématique des *caches* constitue à elle seule un axe de recherche complet et complexe. Nous avons choisi de présenter d'une manière très synthétique les techniques majeures mises en œuvre en milieu distribué (cf. tableau 2.1). Chaque approche décrite au sein du tableau insiste sur des critères spécifiques et parfois très distincts ; l'intérêt n'est pas de regrouper les méthodes afin de les comparer mais d'informer le lecteur sur les approches fondamentales et la terminologie qui s'y rapporte. Si l'utilisation de gestionnaires de cache permet en effet de réduire les interactions avec le système de stockage, il ne permet pas de prendre directement en compte les contraintes des E/S parallèles lors du premier accès au support physique de stockage. C'est pour cela que nous n'avons pas souhaité détailler les différentes techniques.

Méthode	**Description**
Client to Client transfers	Exploitation des *caches* clients distants Les serveurs de données redirigent les requêtes vers le client qui possède déjà une image des données (la charge est ainsi mieux répartie sur la totalité de la grappe). Un système de fichiers [CFKL96] descendant du projet *SPRITE* s'appuie notamment sur ce type d'approche. Une solution plus récente s'appuie également sur ce concept [LCC+05], elle propose de mettre en place un *cache* commun aux processus d'une même application parallélisée.
Cache local à deux niveaux	Le gestionnaire se scinde en deux sous-modules. Le premier, intégré au niveau du noyau, répartit l'espace disponible entre les différents processus en appliquant une méthode de type *LRU*. Le second sous-système, inclus généralement au sein des bibliothèques d'optimisations de plus haut niveau, offre une gestion plus fine de la partie d'espace allouée (*Prefetch*, ...).

Méthode	Description
Cache local vs *Cache* global	Une nouvelle entité est intégrée au système de partage de données afin d'assurer les services de *cache*. Plusieurs modèles ont été suggérés ; le serveur de données transmet par exemple une copie à un nœud de l'architecture qui centralisera l'ensemble des requêtes en provenance des clients. Une seconde approche consiste à laisser les opérations de maintenance du *cache* à la charge des clients ; l'unité centralisée ayant uniquement à sa charge, les opérations de localisation des différentes images (le serveur de données redirige la requête au serveur de *cache* qui va indiquer au client ou retrouver la donnée).
Cache hiérarchique	Mise en place de plusieurs gestionnaires intervenant à plusieurs niveaux au sein de l'architecture (*cache* local sur les clients, global par des entités intermédiaires et enfin *cache* présent sur les serveurs de données et/ou les unités de stockage). Il est important de remarquer que la mise en place de plusieurs tampons à de multiples niveaux peut engendrer dans le pire des cas un surcoût non négligeable. En effet, si chaque interrogation (émission de la requête + recherche au sein du tampon) émet un défaut de page (*cache miss*), l'opération d'E/S sera finalement exécutée. Le coût total se composera alors du temps d'accès au disque additionnée à la latence des caches. Une solution [VSK+03] permet de définir le meilleur «chemin» en outre passant certains niveaux le cas échéant.

TAB. 2.1: Structures de *cache* exploitées en milieu réparti

D'une manière générale, les algorithmes qui composent les gestionnaires de caches se complexifient de manière exponentielle lorsque ceux-ci sont distribués sur l'ensemble de l'architecture et qu'ils doivent prendre en considération des demandes provenant de plusieurs processus (recouvrement, «faux-partage», intégrité entre les multiples copies, ...).

Parallèlement,l'apport de connaissance, comme dans le cas des opérations collectives, pourrait permettre au gestionnaire de cache d'opter pour une stratégie optimale en tenant compte, d'une part, de ces paramètres physiques, et d'autre part, du modèle d'accès en provenance de l'application. Il semble primordial pour les nouveaux systèmes de proposer des politiques de caches fiables et efficaces ; un compromis entre gain, coût (et complexité) doit être clairement défini lors de la mise en place de tels modules.

2.2.4. Stratégies d'ordonnancement

La mise en place de stratégies d'ordonnancement constitue la dernière classe des techniques proposées afin d'améliorer les performances au sein des applications parallèles. Plusieurs travaux théoriques ont étudié différentes stratégies afin de définir les plus efficaces.

Les algorithmes proposés se divisent en deux catégories :

- Les politiques bas niveaux [SCO90] qui ont été abordées au chapitre précédent. Elles tentent de limiter les déplacements des têtes de lectures/écritures au niveau des disques
- Les stratégies d'E/S parallèles [CM01] [JSWB97] : elles consistent à répartir d'une manière optimale les accès sur les différents serveurs d'E/S dans le but de les traiter en parallèle et de minimiser ainsi les temps de réponse.

Si plusieurs de ces solutions ont été mises en œuvre à différents niveaux de la pile des E/S, peu ont été réellement intégrées dans des systèmes de partage de données réparties.

Une stratégie dynamique a été proposée dans le cadre du développement du système de fichiers *PVFS* [Ros00] (cf. section 3.1.6). Cette approche dite réactive (*reactive scheduling*) tente de proposer un ordonnancement adéquat selon la charge et le comportement des systèmes de stockage (côté serveur). 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. La méthode d'ordonnancement réactif est fondée sur une sélection dynamique de politiques d'ordonnancement. Elle utilise les informations sur l'état et la charge du système pour déterminer un ordre d'exécution des requêtes.

Dans le système *Clusterfile* [IMO+04], une heuristique a été également proposée afin de maximiser l'utilisation de chaque serveur de données.

2.3. Bilan

Nous avons décrit les modes d'accès des E/S généralement mis en œuvre dans les applications parallèles. Tout d'abord, la quantité de données traitée est soit négligeable, dans ce cas l'application est surtout dépendante des ressources *CPU*, soit importante et les performances du système de stockage sous-jacent définissent celles de l'application.

La notion d'«importance» est relativement vague et la caractérisation des E/S nous a permis de mieux appréhender les contraintes. Par exemple, une application qui accède à une très grande quantité de données en début d'exécution (comportement compulsif) dépend autant des E/S qu'une application qui manipule des faibles quantités tout au long de son exécution. D'un point de vue global, le rôle des E/S dans ces deux programmmes est primordial.

La principale difficulté dans la gestion des E/S parallèles est de pouvoir servir un nombre conséquent d'accès réalisé au même moment sur un même fichier. Différentes techniques ont été proposées afin de fournir des méthodes d'accès permettant de réduire les dégradations engendrées par de tels comportements. Parmi les classes de solutions disponibles, les techniques visant à agréger les accès sont celles qui ont été le plus étudiées. Ainsi un grand nombre de solutions allant des couches les plus basses aux plus hautes sur les nœuds clients ou bien côté serveur, est disponible. Cependant, les mécanismes sur lesquels elles reposent sont coûteux et ne sont pas toujours efficaces.

Les approches asynchrones ont permis de compléter ces solutions. Toutefois et là encore, leur mise en place au sein d'un code n'est pas toujours triviale, celui-ci étant déjà largement complexifié par l'utilisation d'interfaces spécifiques permettant de définir les masques d'accès propres à chacun des processus.

Les gestionnaires de caches semblent être une approche très efficace. Néanmoins, le maintien de la cohérence des données entre les différents clients nécessite des protocoles spécifiques et rend leur mise en œuvre souvent délicate. De plus, les techniques de pré-chargement utilisées ne sont pas appropriées au comportement des applications parallèles qui s'appuient majoritairement sur des accès de type *strided*. Enfin, quelque soit la mise en œuvre d'un gestionnaire de cache, il sera toujours nécessaire d'accéder au moins une fois à l'unité de stockage. Ce genre de solution doit être, de notre point de vue, considéré comme un complément.

Plusieurs solutions tentent de combiner ces différentes stratégies, elles peuvent être classées en deux grandes catégories : les systèmes de fichiers spécifiquement conçus pour les E/S parallèles et les bibliothèques applicatives.

Les systèmes appartenant à la première catégorie sont souvent trop spécifiques et ont été remplacés par de nouveaux systèmes de fichiers plus génériques. Dans tous les cas, ces systèmes de fichiers «parallélisés» exploitent certains concepts présents au sein des technologies *RAID*[15] afin de réduire les points de contention (goulets d'étranglement). D'une manière générale, ces systèmes tentent de résoudre un large éventail de contraintes comme la gestion de la cohérence, la tolérance aux pannes, la disponibilité, ... et la performance, cette dernière étant un objectif parmi les autres.

Les solutions de la seconde catégorie s'appuient sur les systèmes déjà présents et les complètent par l'intermédiaire de bibliothèques de plus haut niveau (solutions applicatives). Chacune de ces deux catégories comporte des inconvénients (dépendances vis à vis des couches internes, par rapport aux matériels, ...) et des avantages (portabilité, optimisation bas niveau, ...).

Nous allons présenter plusieurs de ces solutions dans le prochain chapitre.

[15] La méthode du *file stripping* (*RAID* 0) est majoritairement employée (cf. section 1.1.6).

SOLUTIONS DISPONIBLES 3

L'enthousiasme pour le développement des *CPUs* et celui des réseaux durant les deux dernières décennies a largement contribué à renforcer la disparité de performance qu'il existe avec les systèmes de stockage. Ainsi, au milieu des années 90, l'éclosion de multiples systèmes massivement parallèles a dévoilé l'insuffisance des systèmes de partage de données disponibles à l'époque (inappropriés d'un point de vue des débits bien sûr, mais également d'un point de vue des interfaces d'utilisation).

De nombreux projets ont vu le jour. Dans un premier temps, ils ont eu pour but la caractérisation du comportement des applications scientifiques *HPC* et la proposition de plusieurs techniques d'optimisation. Nous avons abordé ces aspects dans le chapitre précédent.

S'appuyant sur ces travaux, plusieurs solutions ont été mises en œuvre. Elles peuvent être classées dans deux catégories : les systèmes de fichiers dédiés aux E/S parallèles [CFP+95, MS94, NK97] ou plus «génériques» [CLRT00, Sch03, Sch02, IT03] et les bibliothèques [CBH+94, SCJ+95, BBS+94, MCFX97, Raj02, TGL99a] traitant uniquement la problèmatique des accès parallèles.

Plusieurs ouvrages fournissent une description plus ou moins détaillée de l'ensemble des systèmes disponibles [Sto98, HK04, May01]. Dans ce chapitre, nous avons choisi de présenter les plus représentatifs des différents techniques disponibles. Nous donnerons également une description du système de fichiers `NFS` et des dernières optimisations qui lui ont été apportées [edi03]. En effet, même si ce système de fichiers n'est pas réellement adapté aux architectures dédiées au calcul intensif, il reste un standard et la solution mise en œuvre dans la majeure partie des cas.

3.1. 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 partagé. 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).

Les solutions mises en œuvre s'appuient soit sur un protocole à base d'échange de messages, soit sur la possibilité d'accéder directement à l'ensemble des disques (cf. figure 3.1) :

- 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 : les solutions basées sur ce modèle exploitent des protocoles intégrés aux baies de disques actuelles (*SCSI*, fibre optique) permettant à plusieurs machines d'accéder en concurrence aux diverses données stockées [1] dans le système. Plusieurs machines peuvent accéder en concurrence aux périphériques de stockage. Le système de fichiers *GPFS* (*Global Parallel File System* [Sch02]) s'appuie sur ce modèle.

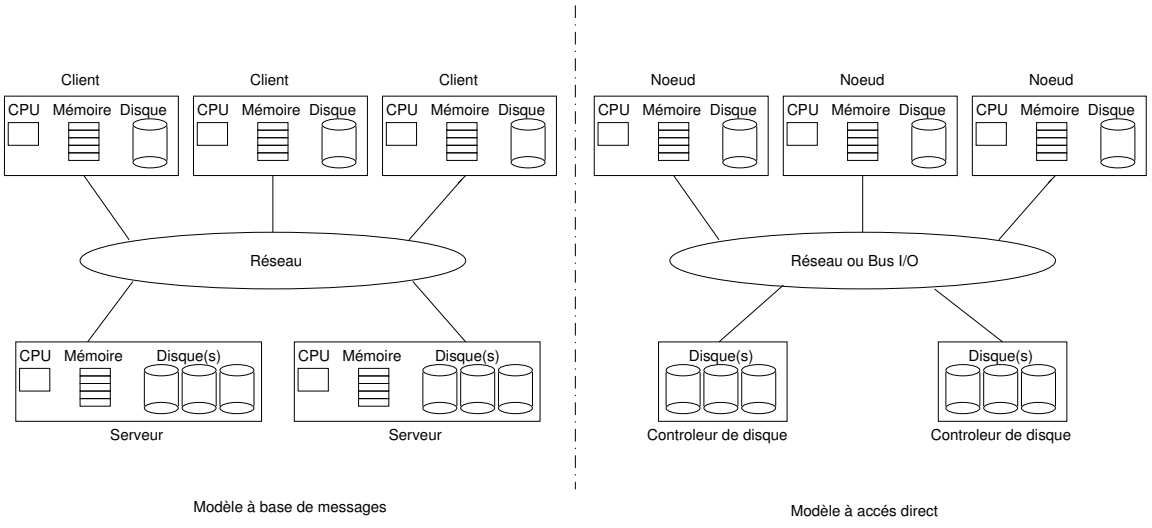


FIG. 3.1 – Protocole de partage de données

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

Les critères de performance (débit, portabilité, granularité de partage, ...) sont influencés par l'approche choisie et les caractéristiques inhérentes aux matériels employés. Toutefois, les solutions à accès direct restent peu déployées car elles sont relativement onéreuses.

En s'appuyant sur le concept de répartition des données sur plusieurs unités de stockage, de nouveaux systèmes, communément appelés, systèmes de fichiers «parallèles» ont été proposés. Grâce à la répartition, plusieurs accès peuvent être traités simultanément. Les nœuds qui sont en charge de gérer les disques sont appelés des nœuds d'E/S (ou serveur d'E/S), les autres nœuds étant des nœuds de calcul.

Un des principaux avantages de cette approche est une utilisation plus fine de l'architecture matérielle sous-jacente. En effet, les couches qui composent ces systèmes ont été pour certaines entièrement re-développées. Ainsi, il est possible de contrôler la totalité des interactions et d'exécuter divers types d'optimisation (agrégation, cache, ...) sur l'ensemble de la pile d'E/S (coté client et coté serveur d'E/S).

A coté de ces optimisations, les systèmes de fichiers parallèles fournissent généralement une interface spécifique à la programmation des E/S parallèles plus ou moins élaborée (stratégie collective, accès recouvrant, ...). Ces routines, qui viennent parfaire dans la majorité des cas les opérations `POSIX` usuelles, sont pour la plupart très compétitives mais restent foncièrement dépendante du modèle déployé. La portabilité est un critère qui fait relativement défaut à ce type de solutions.

Dans un premier temps, nous présentons le système `NFS` et ses dernières évolutions. Comme nous l'avons dit, ce système n'a pas été conçu suivant des critères de performance. Néanmoins, il reste le standard pour partager les données en milieu distribué et est largement déployé sur des petites et moyennes configurations (< 200 nœuds). Par la suite, nous présentons des systèmes plus spécifiques conçus autour du concept de la parallélisation des entités de stockage.

3.1.1. *Network File system*

Le système de fichiers `NFS` est développé par *Sun Microsystems* depuis 1985. Fondé sur une approche client/serveur centralisée (figure 3.2), les clients et le serveur communiquent via le réseau en utilisant plusieurs protocoles définis dans la norme `NFS` et spécifiés au sein du protocole *SunRPC* (*Remote Procedure Calls*). L'ensemble des services est fourni par différents processus démons, qui constituent le serveur ; ils sont accessibles par l'intermédiaire du protocole *mount* et du protocole `NFS`.

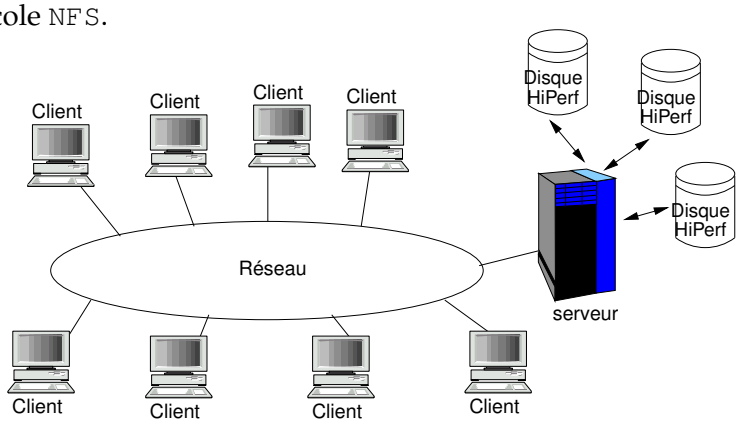


FIG. 3.2 – Architecture `NFS`

Le protocole *mount* : il est utilisé pour établir la connexion initiale entre le client (qui effectue l'appel) et le serveur (qui propose le service). Le serveur maintient une liste de l'ensemble des systèmes de fichiers qu'il exporte ainsi que les nœuds qui ont le droit de les monter. Lorsqu'un client tente de monter un système de fichier distant, le serveur après authentification, lui retourne un descripteur sur le système de fichier. Celui-ci permettra au serveur de résoudre les futures requêtes `NFS` envoyées par le client

Le protocole *NFS* : il fournit un ensemble de procédures (`RPC`) pour réaliser des opérations à distance sur les ressources (recherche, manipulation des liens et des répertoires, gestion des attributs et enfin lecture et écriture sur des fichiers). Chaque appel `RPC` est synchrone au sein du modèle `NFS` ce qui entraîne une mise en veille des clients à chaque appel. Les E/S sur le disque sont réellement exécutées avant de retourner la réponse au client. Le protocole `NFS` ne fournit pas de mécanisme permettant des accès concurrents. Un sous-module (service de verrouillage) a été rajouté, par la suite, pour permettre d'y parvenir mais son utilisation doit être explicite.

Différentes versions ont été et sont développées, la première version de `NFS` est restée confinée au sein des laboratoires de SUN. La version 2 a été mise en place et livrée avec le système d'exploitation *SunOS* 2.01. Depuis cette date (courant 1985), *NFS* est devenu un standard [Edi89]. Actuellement, la version la plus déployée en production est la version 3 [Edi95]. Cette version a apporté principalement des optimisations de performance en écriture en proposant un mode dit «asynchrone» : dans la version 2, toutes les opérations d'écriture sont sérialisées, c'est-à-dire, que le service côté client (démon `rpciod`) attend un acquittement du serveur avant de transmettre la prochaine requête. Depuis la version 3, le démon peut transmettre plusieurs requêtes puis demander un acquittement pour l'ensemble des requêtes qu'il a transmises.

La dernière version disponible est la version 4. Même si cette version est officiellement fournie dans le noyau *Linux 2.6.12*, elle reste encore peu mise en œuvre. Les principales améliorations concernent :

- la gestion de la sécurité via le système d'authentification sécurisé (kerberos5) ;
- l'ajout de fonctionnalités permettant de déplacer/répliquer un serveur `NFS` d'une machine à une autre de manière transparente ;
- une meilleure disponibilité avec une reprise sur incidents ;
- un meilleur passage à l'échelle en réduisant le traffic par groupement de requêtes et l'utilisation du concept de délégation.

Cette nouvelle version marque une rupture totale avec les versions précédentes. Ces nouvelles évolutions la rendent incompatible avec les versions précédentes. Par exemple, la reprise sur incident et la délégation impliquent que `NFS` v4 soit un serveur à état (*statefull*) à l'opposé du modèle *stateless* utilisé dans les versions précédentes. De plus `NFS` v4 ne repose plus que sur le protocole `TCP`.

Plusieurs versions parallélisées de `NFS` ont été proposées. Actuellement, une version [2] est développée au sein du laboratoire ID-IMAG situé à Grenoble. L'idée principale est d'offrir aux clients de manière transparente une gestion parallélisée des accès [LDVL03]. Une autre solution [HH05] fondée sur l'architecture *PVFS* est également disponible. Cependant, cette version est plus intrusive car elle nécessite que les clients et le serveur utilisent un code spécifique.

Dans les paragraphes suivants, nous allons décrire plusieurs systèmes de fichiers «parallèles», d'abord spécifiques à la gestion des E/S parallèles puis plus génériques.

[2] `http://nfsp.imag.fr`

3.1.2. De *Vesta* à *PIOFS*

Vesta

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

La première notion importante relative à ce système est le placement physique des données sur une base à 2 niveaux : un fichier est divisé en un nombre de cellules (partitionnement physique) spécifié lors de sa création, chacune de ces cellules étant elle-même subdivisée en blocs de taille paramétrable (comparable aux blocs fichiers, cf. 1.2.2.3). Les tailles de répartition des données étant positionnées lors de la création, il est alors possible de choisir pour chacun des fichiers une granularité permettant de minimiser les accès disque et de ce fait, obtenir de meilleures performances.

Le second concept est la mise à disposition d'une interface permettant de créer des vues logiques pour accéder aux données. Cette notion, entièrement reprise par la suite dans le standard `MPI I/O`, offre au programmeur un accès transparent à des informations plus ou moins diffuses.

L'exploitation d'un cache intégré aux serveurs d'E/S permet d'optimiser les accès en lecture et en écriture. La cohérence est alors maintenue au sein du système via un algorithme à base de jeton [3] . Les opérations d'écriture réparties sur plusieurs serveurs de stockage reposent également sur cette méthode afin d'assurer la consistance requise.

Vesta est un système qui a pris en compte plusieurs exigences de la programmation des E/S parallèles. Néanmoins, il a été choisi de ne pas intégrer l'interface `POSIX` ce qui a réduit considérablement son essor. De même, aucune optimisation des opérations collectives n'est incluse. La gestion du cache du côté client est totalement absente.

3.1.3. *Parallel I/O File System*

Le système PIOFS [CFP+95] s'insère dans la continuité des travaux entrepris lors du projet *Vesta*. Intégré au système d'exploitation *AIX* d'IBM, il fournit en plus des fonctionnalités proposées par le modèle précédent, une partie de l'interface `POSIX` [4] indisponible auparavant. Toutefois, cet apport primordial n'a pas réellement accru la portabilité du modèle. En effet, l'appel aux routines optimisées d'accès parallèles reste explicite (et par conséquent figure toujours au sein du programme).

Un des avantages de *PIOFS* est la mise à disposition d'outils permettant le suivi des applications. Ces fonctionnalités (trace et *Profilling*) se révèlent très utiles dans le cadre du développement d'applications scientifiques hautement parallèles [5] .

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

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

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

3.1.4. *Parallel Input/OUtput System*

L'architecture *PIOUS* [6] , présentée courant 1993, a introduit le concept de groupe au sein des opérations d'E/S parallèles [MS94]. En plus des vues logiques, nommées *para-fichiers* et comparables aux cellules dans *Vesta*, il suggère trois modes d'accès parallèle au sein d'un ensemble spécifié de nœuds (notion de groupe partageant une même vue logique d'une ressource) :

1. vue globale, le *para-fichier* apparaît comme une séquence linéaire d'octets. Chaque unité concernée partage un pointeur commun (cf. paragraphe 2.1.2) sur cette ressource (toute opération est atomique).
2. vue indépendante, à la différence du mode global, un pointeur est attribué à chaque processus.
3. vue segmentée, chaque nœud du groupe accède de manière atomique à un ou plusieurs segments indépendants du *para-fichier*.

Le modèle *PIOUS* fournit également un service de verrou avec une granularité correspondant à l'octet (le terme de *Byte-Range locking* est généralement employé).

L'architecture globale se compose d'un service de coordination (*PSC*) contenant les méta-informations et l'état du système, d'un ensemble de serveurs de données (*PDS*) et d'une bibliothèque de routines d'accès parallèles à lier avec les applications clientes. La particularité de ce système est l'utilisation de systèmes sous-jacents. Les serveurs (*PSC* et *PDC*) s'appuient sur des systèmes de fichiers natifs (*UNIX*, *XFS*, ...) pour stocker de manière pérenne les données. Par ailleurs, un mécanisme de transport pour l'échange des messages entre les entités est requis. Ce système repose sur l'environnement de programmation *PVM* (*Parallel Virtual Machine*) ce qui limite donc son utilisation à cet environnement.

3.1.5. *Galley File System*

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

Une des toutes premières distinctions du système *Galley* part du constat qu'une distribution statique des fichiers sur n unités de stockage peut se révéler moins performante que l'approche linéaire des systèmes de fichiers standard [7] . Ainsi la répartition des données sur l'ensemble des unités de stockage, appelé *IOP*, n'est pas imposée (tout fichier de taille quelconque peut être sauvegardé sur un unique disque).

S'appuyant sur une approche client/serveur (unité de calcul *CP* et nœud de stockage *IOP*), les processus de calcul sont indépendants entre-eux et ne nécessitent pas un intergiciel de communication évoluée de type `PVM` ou `MPI` comme il est requis au sein de *PIOFS*. S'appuyant sur

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

[7] En effet, les requêtes de taille supérieure au coefficient de répartition (taille d'un *chunk*) génèrent une gestion plus complexe et une latence d'E/S plus élevée.

le multi-tâche et les couches TCP/IP, il intègre les opérations asynchrones le cas échéant et utilise un gestionnaire de cache sur les serveurs de données afin d'optimiser les opérations d'E/S. Un gestionnaire de disque permet d'interfacer le système de fichiers natif installé sur l'*IOP*.

Une fois que le client a chargé la partie cliente du système *Galley*, il peut utiliser le système de partage de données. En plus des traditionnelles routines `POSIX`, des appels plus complexes fournissent un contrôle sur le placement physique des données lors de la création des fichiers mais également la possibilité de réaliser des accès recouvrants (cf. section 2.1.4).

Une modélisation proche de celles présentées dans les architectures précédentes est mise en place afin d'abstraire les fichiers et de proposer des vues logiques. Ces derniers sont composés d'un ou plusieurs «sous-fichier(s)» sauvegardé(s) chacun sur une unité de stockage à part. Chaque sous-fichier est constitué d'un ensemble de taille variable de *fork* (composant comparable à un fichier traditionnel `UNIX` mais nécessitant moins d'opérations au niveau des méta-informations).

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

3.1.6. *Parallel Virtual File System*

Le système de fichier *PVFS* a été introduit courant 1996, [LR96]. Il a pour principal objectif l'exploitation optimisée des ressources présentes au sein des grappes (bande passante, espace disque disponible, ...). Fondé sur une architecture client/serveur, une fois encore, le système se décompose en un méta-serveur (*mgr*, stockant les méta-informations) et en un groupe de serveurs de données (*iods*, pour les données effectives).
Chaque fichier est découpé en blocs de taille fixe répartis de manière circulaire sur l'ensemble des unités de sauvegarde. Lorsqu'une application désire lire ou écrire des données, les informations concernant la distribution sont d'abord récupérées auprès du *mgr* par la machine cliente. Une fois le découpage connu, les clients accèdent directement aux serveurs de données. Les problèmes conséquents au goulot d'étranglement au niveau du méta-serveur (le *mgr*) sont ainsi largement diminués puisque seules les méta-requêtes stressent le serveur [8] .

Un principe fondamental du modèle est l'agrégation des requêtes d'E/S au sein d'une même trame réseau (*Stream-Based I/O*, cf. section 2.2.1.2). Cette technique permet de réduire considérablement les flux réseaux et diminue ainsi les phénomènes de contention.

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

L'environnement *PVFS* propose trois interfaces d'accès. Les routines `POSIX` sont fournies en utilisant la couche *VFS*. Une bibliothèque pour les opérations sur des données disjointes vient parfaire l'interface `UNIX` (définition de vues logiques appelées *MDBI* [9]). Enfin, une mise en œuvre d'un pilote spécifique pour les couches basses de la bibliothèque *ROMIO* a été réalisée.

[8] A la différence des modèles à la `NFS` où la totalité des requêtes transite par le point centralisé.
[9] Multi Dimensional Block Interface

Le modèle *PVFS* a été relativement accepté par la communauté *HPC*. Plusieurs travaux ont suggéré diverses optimisations. Par exemple un module noyau proposant un service de cache a été mis en œuvre [VSK+03]. Cependant, l'absence de certaines fonctionnalités fondamentales comme la tolérance aux pannes (dépendance entre les serveurs de données), l'inexistence de verrous ou encore la distribution figée des informations ne permettent pas une implantation unanime du modèle.

Depuis début 2005, une seconde version, *PVFS 2* est disponible [LMRC04]. En cours de finalisation, cette version a été proposée afin de corriger et d'éliminer les divers inconvénients. Tout d'abord, l'ensemble des serveurs est susceptible de gérer tout type de requête afin d'éliminer toute présence de point critique (le méta-serveur est distribué sur l'ensemble des nœuds de stockage par un système de *hash*). Une couche fournissant le standard `MPI I/O` a été développée au-dessus du système. Elle permet d'accèder aux données non contiguës reposant sur des modèles à régularité double, triple, ...(cf. section 2.1.4). Un module d'opérations asynchrones devrait également être fourni. Pour terminer, comme l'a suggéré le projet *Galley*, la nouvelle structure a été développée de manière à pouvoir lier rapidement des modules pour des caractéristiques spécifiques (distribution de données, ordonnancement, protocole de transport).

3.1.7. *Clusterfile*

Développé à l'université de Karlsruhe, le système de fichiers *Clusterfile* [IT03] [IMO+04] exploite plusieurs serveurs d'E/S. Chaque fichier se compose de sous-fichiers distribués sur un ou plusieurs serveurs.

Le système *Clusterfile* a été développé principalement pour les E/S parallèles, il est constitué de trois composants principaux :

- Un composant de gestion de méta-données : il maintient à jour les informations concernant la structure des fichiers ainsi que la distribution physique des sous-fichiers sur les différents disques.
- Plusieurs serveurs d'E/S : ils 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 bibliothèque d'E/S : elle est utilisé par les nœuds de calcul pour exécuter des opérations d'E/S sur le système de fichiers. Elle fournit une interface enrichie en plus des routines standard `UNIX` pour les utilisateurs.

Le système Clusterfile permet aux utilisateurs de spécifier leurs accès par des vues (*View I/O*, similaires aux vues utilisées au sein des bases de données). Une vue est une séquence d'adresses contiguës d'octets, elle est composée par une ou plusieurs régions d'un fichier. La création des vues se fait par l'appel de routines spécifiques au sein du code de l'application parallèle. Dès qu'une vue est créée côté client, elle est transmise aux serveurs d'E/S. Les serveurs d'E/S utilisent ces informations afin de prévoir les accès et donc 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.

3.1.8. *Lustre*

Lustre [10] , [Inc02, Sch03], est une solution en cours de développement et a pour objectif la mise en œuvre d'un système de fichiers adapté pour les architectures de type grappe [11] . La brève description qui suit n'insistera pas réellement sur les aspects parallèles et les techniques mises en œuvre afin d'en optimiser les accès (l'architecture étant encore très évolutive). Toutefois, le modèle agrégeant de nombreux concepts suggérés par la communauté scientifique depuis plus d'une quinzaine d'années [12] [Bra99] semble être très prometteur (partage à grain fin, fiabilité, portabilité, passage à l'échelle, ...). Son déploiement sur plusieurs gros supercalculateurs ainsi que le soutien apporté par de multiples partenaires privés ou universitaires nous incite à prendre en compte ce modèle afin d'observer comment les problématiques liées aux accès parallèles au sein des applications *HPC* pourront être organisées.

L'architecture est globalement comparable à la première version de *PVFS*. Un méta-serveur, *MDS*, sauvegarde l'état du système de fichiers (les méta-informations) et des serveurs de données (*OST*, *Object Storage Target*) gèrent l'ensemble des requêtes d'E/S. Afin de réduire les inconvénients engendrés par un point centralisé, le *MDS* est répliqué (une image «miroir» est réalisée mais non activée) et intervient uniquement lors d'opérations modifiant les méta-informations (création, destruction et mise à jour).

Le stockage des fichiers est fondé sur les concepts de «bloc objet» (*object based disk* [MGR03], cf. section 1.2.2.3). Cependant, *Lustre* fournit plusieurs pilotes permettant d'exporter des systèmes de fichiers locaux comme *Ext3*, *ReiserFS* ou encore *XFS* et donner ainsi la possibilité au modèle de s'adapter à plusieurs types d'architectures.

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

Une autre caractéristique de *Lustre* est son indépendance vis-à-vis du réseau et des protocoles utilisés grâce à l'utilisation de la bibliothèque *Portal* [13] . Une couche basse du modèle, *NAL* (*Network Abstraction Layer*), fournit une interface commune à de multiples protocoles réseaux. Actuellement, le système supporte les flux *TCP/IP*, les réseaux *Quadrics*, *Myrinet* ou encore *SCI* (d'autres pilotes, comme *InfiniBand* notamment, sont en phase de développement). Ce détachement permet au système tout d'abord de bénéficier des apports réalisés à tout moment au sein des protocoles réseaux mais par ailleurs de coupler ces différents supports au sein d'une même et unique architecture (certains clients accèdent aux différents serveurs par l'intermédiaire d'un réseau *TCP* et d'autres par un réseau *Quadrics* par exemple). Le protocole *Lustre* offre également la possibilité de court-circuiter les piles réseaux au cas où un même nœud inclût le processus client et le processus *OST*.

[10] `http://www.lustre.org`

[11] L'acronyme *Lustre* provient de l'association de *Linux* et de *cluster*.

[12] Le modèle est un descendant du système *VAX Cluster* et exploite plusieurs notions également utilisées dans le plus récent *GPFS*.

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

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

Dans un premier temps, les clients ont exploité un cache permettant de réaliser des écritures «retardées» sur les méta-informations. Cette couche a été étendue vers les concepts de cache persistant proposés au sein de systèmes de fichiers comme *AFS*, *CODA* ou encore *Intermezzo*, [Leb02]. Donnant la possibilité de travailler en mode déconnecté dans les systèmes de fichiers précédemment cités, le module mis en place est utilisé ici afin de réduire les flux réseaux : un «méta-serveur local» est ajouté sur chaque client, une image cohérente des méta-informations est ainsi disponible ponctuellement. Cette image est propagée vers le *MDS* distant lors des requêtes ne pouvant être satisfaites localement (cache hiérarchique).

Parallèlement au gestionnaire de cache des méta-données, un mécanisme de cache «global» a été mis en place pour optimiser les accès directs vers les *OSTs*. Ces serveurs indépendants déployés sur des nœuds distants récupèrent une copie de la portion de données souhaitée. Les clients qui émettent des accès vers ces données, sont redirigés par l'*OST* concerné, vers l'image temporairement stockée sur le gestionnaire distant. Cette approche réduit les phénomènes de saturation pouvant intervenir sur un serveur de stockage lorsqu'un trop grand nombre de clients tente d'accéder à des données présentes au sein d'un même *OST*.

Une couche permettant d'interfacer la bibliothèque `MPI I/O` est également en cours de finalisation. Un des buts, est par exemple, une synchronisation des opérations collectives au niveau des *OST*.

Le système de fichiers *Lustre* est en phase de développement intensif depuis un peu plus de 3 ans maintenant. Des versions intermédiaires sont disponibles sous les systèmes *Red Hat Enterprise Linux Advanced Server*, *BULL advanced server* et *SUSE Linux Entreprise Server* qui exploitent des noyaux *Linux* particuliers. Il est aujourd'hui utilisé par plus de 50 architectures figurant au TOP 500. Cependant, sa mise en œuvre sous les noyaux *Linux* standard nécessite la modification de ce dernier et reste une opération fastidieuse à réaliser.

3.1.9. Bilan

Les premiers systèmes de fichiers «parallèles» ont été conçus afin d'optimiser les critères de performance et diminuer l'impact subi par les applications parallélisées fortement dépendantes des E/S. Plusieurs routines ont enrichi l'interface standard `POSIX` dans le but de gérer de manière plus fine le placement physique des données. Le principal but consiste à attribuer une granularité de découpage en adéquation avec les schémas d'accès exploités par les applications. Un accès manipulant une taille juste un peu plus grande que la taille de découpage (*chunk*, unité élémentaire de répartition) demande un coût de gestion plus important et se révèle donc inefficace. Malheureusement la complexité de l'ensemble de ces interfaces a été un élément bloquant pour leur acceptation. Certaines des solutions ont développé des couches d'abstraction afin de simplifier les nombreuses routines mais en ayant évidemment un impact au niveau des performances.

Depuis la fin des années 90, les systèmes de fichiers parallèles proposés prennent en compte un plus grand nombre de facteurs comme la cohérence entre les données, la granularité d'accès, la tolérance aux pannes, le facteur de passage à l'échelle ou encore des aspects liés à la sécurité. La résolution de l'ensemble de ces paramètres complexifient leur mise en œuvre en plus d'imposer des compromis, les performances passant parfois en second plan.

Caractéristique \ Système		*NFS*	*Vesta/PIOFS*	*PIOUS*	*Galley*	*PVFS*	*ClusterFile*	*Lustre*
Stockage	**Architecture**	Centralisé	Parallèle	Parallèle	Parallèle	Parallèle	Parallèle	Parallèle
	Abstractions des fichiers	Fichier (bloc de taille fixe)	Fichier à deux niveaux (cellule et bloc de taille paramétrable à la création du fichier)	Fichier à deux niveaux (*parafichier*, notion de groupe et descripteur partagé)	Fichier à trois niveaux, *striping* paramétrable	*Striping* fixe (*round robin*), paramétrable depuis la version 2	Définition des vues pour le stockage	*Striping* paramètrable (concept de disque objet)
Fonctionnalités	**Interface `POSIX`**	OUI	Absent dans Vesta	OUI	L'interface est très riche mais complexe. Les API PIOFS et Panda, plus simple d'utilisation, ont été portées au dessus.	OUI	OUI	OUI
	Définition de vues	NON	OUI	OUI		*Stream Based I/O*	OUI	via `MPI I/O`
	Accès collectifs	NON	OUI	??		via `MPI I/O`	OUI	via `MPI I/O`
	accès à grains fins	à partir de la version 4	cohérence à base de jeton	*Byte Range Locking*		Pas de verrous	??	*Byte Range Locking*
	mode asynchrone	via la couche `aio`	OUI	OUI		Prévue mais non disponible	??	via la couche `aio`
	API spéficique	NON	OUI	OUI		OUI	OUI	??
	Optimisation `MPI I/O`	NON	OUI	NON	NON	OUI	NON	OUI
	Gestionnaire de cache	Côté client, utilisation du cache natif côté serveur	Côté serveur	Utilisation du cache natif côté serveur	Côté serveur	Cache global + utilisation du cache natif côté serveur	Cache global	Cache global
Prérequis	***OS***	`UNIX`-*Like*	`AIX` (*IBM*)	`UNIX`-*Like*	`UNIX`-*Like*	*Linux*	*Linux*	*Linux*
	Divers			`PVM`	`TCP/IP`		`TCP I/P`	
Utilisation		Largement déployé	Abandonné peu à peu	??	Pas à notre connaissance	OUI	Pas de version de production	Déployé sur des grosses configurations

TAB. 3.1 – Récapitulatif des caractéristiques des systèmes présentés

Dans la suite du document, nous décrivons plusieurs bibliothèques dédiées aux E/S parallèles. Ces solutions applicatives peuvent justement venir compléter les systèmes de fichiers «nouvelle-génération» et apporter les fonctionnalités nécessaires pour la gestion efficace des E/S parallèles.

3.2. Les bibliothèques d'E/S

Généralement moins intrusive du point de vue des systèmes d'exploitation, les bibliothèques d'E/S parallèles fournissent aux utilisateurs des interfaces de programmation de haut niveau. Le principal but de ces solutions consiste à intégrer plusieurs techniques d'optimisation afin d'améliorer les performances tout en se détachant de l'architecture de stockage sous-jacente [14] (critère de portabilité). Par ailleurs, leur mise en œuvre généralement réalisée en espace utilisateur permet d'intervenir au sein des diverses couches afin d'apporter d'éventuelles modifications spécifiques à certaines applications ou également de compléter les interfaces avec de nouvelles politiques d'optimisation. Après plusieurs solutions et dans un souci d'uniformisation des interfaces, le consortium `MPI` a défini le standard `MPI I/O` [15] . L'implantation la plus déployée de ce standard est la bibliothèque *ROMIO* [TGL99a]. Parallèlement, plusieurs bibliothèques précédemment proposées continuent d'être utilisées. Nous présentons ces différentes solutions dans les paragraphes suivants.

3.2.1. *PASSION*

Le programme *Parallel And Scalable Software for Input-Output* [CBH+94], débuté en 1994, a eu pour principal ambition la mise en place d'un système capable de résoudre les problèmes d'E/S dans les applications *out of core* (cf. section 2.1.3) sur architectures à mémoires distribuées. Une des premières caractéristiques est la mise à disposition d'un support pour la compilation permettant d'optimiser implicitement du code *Fortran 90D* ou bien *HPF*. Les optimisations disponibles sont des techniques de pré-chargement, d'accès recouvrant, d'opération collective ou encore d'ordonnancement optimisé des données, ...) . Ces mécanismes sont également disponibles durant la phase d'exécution par l'intermédiaire de routines spécifiques.

S'appuyant sur un modèle scindé en plusieurs entités (gestionnaire de cache, gestionnaire d'opérations collectives), la bibliothèque peut interagir de manière optimale avec le système de fichiers sous-jacent en exploitant à chaque niveau la connaissance du placement des données. Par ailleurs, de nombreuses fonctions permettant la gestion des calculs *out-of-core* (multiplication de matrices, transformations ...) sont incluses dans la bibliothèque.

Les couches *PASSION* sont utilisables au-dessus des systèmes de fichiers Intel CFS et *Vesta*.

3.2.2. *PANDA*

Introduit en 1994, le projet *PANDA* [16] [SCJ+95] s'est concentré sur les problèmes d'E/S attenant à la gestion des tableaux à plusieurs dimensions. La bibliothèque mise en œuvre a

[14] Au contraire des systèmes de fichiers précédemment présentés, qui sont eux, étroitement liés aussi bien avec leurs couches logicielles qu'avec la configuration matérielle exigée.

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

[16] *Persistence AND Array*.

été conçue dans le cadre d'applications parallèles s'exécutant sur plusieurs nœuds SMP. Un modèle de type client/serveur (*Server-Directed I/O*) a été retenu (cf. section 2.2.1.4). Développée en `C++`, la solution *PANDA* s'appuie sur la bibliothèque d'échange de messages `MPI` et peut être utilisée au-dessus de tous les systèmes de fichiers proposant une interface `UNIX`.

Elle fournit la possibilité de distribuer les informations au sein des divers serveurs de stockage selon une disposition donnée. D'une manière générale, le placement physique des données correspond à celui employé en mémoire.

Une opération collective se déroule de la manière suivante : un client, élu (*master client*) parmi le groupe concerné par l'opération collective, transmet vers un serveur également choisi (*master server*) une description de la disposition des données en mémoire et du placement souhaité sur les disques.

Les serveurs de stockage vont exploiter ces informations afin de transmettre ou récupérer auprès des clients chaque(s) bloc(s) nécessaire à l'exécution de la requête.

Lorsque l'opération d'E/S se termine sur un serveur, celui-ci en avertit le gestionnaire de la requête (*master server*). Une fois que la totalité des serveurs de stockage ont acquitté l'opération, le gestionnaire en fait part au client principal. Ce dernier indiquera le bon déroulement de l'opération à l'ensemble des nœuds clients.

De nombreuses fonctionnalités ont été ajoutées depuis la création du projet. En 1997, la bibliothèque *PANDA* a intégré une interface permettant la migration performante de données entre deux architectures [17] durant l'exécution d'un programme [KWC+97].

Le système a intégré une nouvelle approche appelée *part-time I/O* [CWS+97]. Dans cette méthode, un nœud peut jouer le rôle d'un client et d'un serveur d'E/S. Il peut être un serveur pendant un temps puis reprendre le rôle d'un client après avoir fini les E/S. D'une part, cette technique permet de réduire 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.

En 2002, il intègre une nouvelle optimisation au sein des opérations collectives [MWLY02]. la technique *active buffering* exploite une approche asynchrone afin de rendre transparentes les latences liées aux E/S pour les écritures ; lors d'opérations d'écriture collectives, l'espace mémoire disponible sur chaque serveur est utilisé afin de sauvegarder temporairement les données transmises par les clients. Une fois que chaque serveur a récupéré les données au sein de ces tampons, il valide l'opération au serveur maître qui transmettra l'information aux clients. L'opération effective de l'écriture des données sur disques sera alors réalisée lorsque les serveurs d'E/S seront inactifs.

Le projet *PANDA* est toujours d'actualité. Intégré au sein d'un projet de simulation de fusée, la bibliothèque *RocPANDA* [MJoMW03], se concentre sur les problématiques liées à la compatibilité des données et des accès entre les différentes modules composant une application *HPC* (modules en permanente évolution et qui de plus sont développés par plusieurs groupes de recherche souvent indépendants et programmant sous des langages et selon des règles différentes *C*, *C++*, *HPF*, ...).

[17] Par exemple, d'une grappe vers une station d'analyse indépendante de l'architecture.

3.2.3. *PPFS*

Portable Parallel File System [HER+95] est un système développé au sein de l'équipe *PABLO* [18] . Fondée sur une approche client/serveur, cette bibliothèque permet de contrôler d'une part le placement des données sur les diverses unités de stockage mais également les diverses stratégies d'optimisation implémentées.

Ainsi, *PPFS* fournit :

- une gestion du cache paramétrable de manière globale, au niveau des clients ou des serveurs ;
- des techniques de pré-chargement sur les clients et sur les serveurs ;
- un degré de cohérence, plus ou moins forte entre les méta-informations mais également au sein des données ;
- plusieurs modes d'accès, séquentiel, aléatoire ou structuré pour les données disjointes.

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

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

Comme nous l'avons précédemment indiqué, *PPFS* exploite les techniques de caches à plusieurs niveaux. Le concept d'agent de cache consistant en une unité globale par laquelle les clients peuvent se partager les données d'un fichier offre une répartition des demandes d'E/S entre agents et serveurs de données. La couche «cliente», susceptible également d'intégrer un cache local, interroge le «méta-serveur» afin de déterminer quels serveurs de données ou agents de caches va pouvoir satisfaire sa demande. Toutes les requêtes futures sur ce fichier seront transmises directement vers l'entité spécifiée. L'utilisation de ces agents est dynamique : des routines destinées à la mise en place ainsi qu'à la suppression de tels modules sont incluses au sein de la bibliothèque. Il est, par conséquent, possible d'équilibrer la charge liée aux E/S durant tout le déroulement de l'application *HPC* .

Un autre point intéressant de ce concept est une gestion «paramétrable» de la cohérence. En effet, lorsqu'un agent est instancié, la totalité des accès au fichier associé transite par ce point centralisé et pourra être maintenue en cohérence (l'utilisation de cette méthode s'avère moins coûteuse que la mise en place de caches distribuées).

Récemment, l'équipe *PABLO* a présenté un nouveau modèle. *PPFS II* exploite un logiciel permettant d'analyser «en temps-réel» l'état et les performances de l'architecture sur lequel il est déployé. Les résultats fournis par ce module, nommée *autopilot*, sont utilisés par la bibliothèque afin de réajuster de manière dynamique et transparente les stratégies d'optimisation employées.

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

3.2.4. *Jovian*

En 1994, dans le cadre du projet *CHAOS* [19] , la bibliothèque *Jovian* [BBS+94] a été présentée. Cette bibliothèque permet d'optimiser les accès aux E/S au sein des programmes de type *SPMD* [20] déployés sur des architectures composées de plusieurs unités de stockage. Le concept fondamental est que toute opération d'E/S se déroule de manière collective (tous les nœuds communs à l'application réaliseront l'opération et ceci dans tous les cas).

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

Lors d'une opération d'E/S de type lecture ou écriture, chaque *AP* émet de manière bloquante, sa requête vers un *CP* déterminé lors de la phase d'initialisation. Dès cette première étape, la bibliothèque, côté client, exploite une méthode analogue à l'algorithme *Data Sieving* décrite au chapitre précédent. A la réception des demandes, l'ensemble des *CPs* entre dans une phase de communication collective afin de redistribuer, suivant la portion de fichier requise, les demandes d'E/S au *CP* en charge de cette plage d'octets (attribution des requêtes logiques par rapport au placement physique). Une fois que la totalité des demandes a été correctement redirigée, chaque *CP* exécute, de manière similaire aux *APs*, une méthode d'agrégation des requêtes (*Data Sieving*) afin de maximiser l'utilisation de la bande d'E/S disponible [21] . L'accès «physique» est alors réalisé sur l'unité de stockage correspondante, soit par l'intermédiaire d'un contrôleur permettant d'interfacer directement les baies de disques, soit simplement en s'appuyant sur l'interface `POSIX` du système de fichiers natif. Les données récupérées sont placées au sein de tampons «d'envoi» (un tampon d'envoi par *AP* concerné) ; ces derniers seront transmis lorsque l'opération «physique» sera terminée (ou que le tampon temporaire sera saturé).

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

– *Jovian* s'appuyait sur une approche synchrone des opérations d'E/S : seules des opérations collectives pouvaient apparaître. La nouvelle version a supprimé la totalité des routines collectives et propose uniquement une interface `POSIX` simple permettant par l'intermédiaire de la routine *lio_listio()* d'accéder à des données non contiguës en un unique appel.

[19] Université du Maryland.

[20] *Single Program Multiple Data*.

[21] Le paramètre *Read Gap Size*, similaire au dégré de continuité, est utilisé afin d'éviter les agrégations trop coûteuses, cf. 2.2.1.3.

– Comme il a été indiqué, l'approche employée au sein de *Jovian* consistait en une agrégation des requêtes de manière collective. Ainsi, toute opération d'E/S ne pouvait physiquement être exécutée qu'une fois l'ensemble des «pré-phases» terminées. A la différence de *Jovian*, *Jovian 2* tente de maximiser l'utilisation du disque en retournant de la manière la plus rapide la plage de données demandée.
– Le module *CP* est implémenté au sein d'un *thread* intégré soit à l'application cliente soit sur un nœud distant. L'intérêt est de pouvoir réaliser des architectures «pair à pair» si besoin est. Dans ce cas tout nœud client est susceptible de se comporter comme un serveur pour toute autre machine. Ainsi il est également possible de «court-circuiter» la procédure d'accès aux E/S pour toute demande à destination du *thread* local et éviter les étapes de communication (technique notamment employée au sein de la bibliothèque *PANDA* et reprise lors de la conception du système de fichier *Lustre*).
– Enfin, cette solution offre la possibilité de gérer la distribution du placement physique des données sur les unités de stockage sous-jacentes.

3.2.5. *MTIO*

La bibliothèque *MTIO* [MCFX97], conçue en 1997, s'appuie également sur les concepts multi-tâches dans le but de réduire les latences d'E/S. Elle s'attarde principalement sur l'optimisation des accès individuels ou collectifs. Dans le cadre des requêtes indépendantes, elle exploite le *multi-threading* afin de maximiser l'utilisation du processus.

Lors du lancement du programme, chaque processus, créé sur un nœud, se compose d'un premier *thread* de calcul (*CT*) et d'un second pour les opération d'E/S (*IOT*). L'idée est de réaliser l'appel d'E/S (souvent bloquant sur la quasi-totalité des systèmes de fichiers) dans le processus léger et de pouvoir ainsi exploiter le *CPU* pour exécuter une autre partie du code en attendant les informations retournées une fois la demande exécutée.

La gestion des accès, de la cohérence et du partage des données doit être réalisée par le système de fichiers sous-jacent. La bibliothèque *MTIO*, qui emploie notamment la couche `MPI` pour la communication inter-nœud, fournit uniquement deux modèles d'accès : un modèle d'accès individuel, décrit succinctement auparavant, et un modèle collectif.

La spécificité de ce second mode est la prise en compte de paramètres permettant dans un premier temps, de minimiser la phase de communication inter-processus (phase relativement coûteuse au sein de l'algorithme *Two-Phase*) puis par la suite d'équilibrer au mieux les requêtes sur l'ensemble des nœuds concernés par l'opération collective.

Lors de la première étape, les requêtes sont attribuées selon la localité des données d'un point de vue physique puis d'un point de vue logique. Lorsqu'une requête ne répond spécifiquement pas à un des critères de localité, elle est alors attribuée au *thread* d'E/S qui est pourvu du plus petit nombre d'appels. Un équilibrage des demandes constituant l'opération collective est ainsi fait.

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

3.2.6. Bibliothèques `MPI I/O`

Dans un souci d'uniformisation des interfaces et afin d'améliorer la portabilité des applications scientifiques, les utilisateurs et les industriels ont entrepris la conception d'une interface commune autour du modèle de programmation `MPI`. Ainsi, en 1997, lors de la définition du standard `MPI-2`, le modèle `MPI I/O` a été introduit [CFF+01]. Il définit un ensemble de routines permettant le transfert de données «efficace» depuis et vers les systèmes de fichiers. Le modèle, cependant, ne fournit pas de routines permettant d'intéragir avec le placement physiques des données.

Nous allons présenter deux solutions qui répondent à ce modèle, la première *ROMIO* est la plus connue et la plus déployée, la seconde, *Mercutio* apporte plusieurs améliorations qui nous semblent pertinentes de présenter.

ROMIO

Dans le cadre des travaux initiés par le laboratoire Argonne autour du standard `MPI` et des E/S parallèles, une solution proposant les spécifications `MPI I/O`, intitulée *ROMIO* [22] , a été présentée en octobre 1997. Cette solution est incluse dans les bibliothèques *MPICH* et *LAM* du standard `MPI` et est aujourd'hui la plus déployée. Un grand nombre d'articles fournissent une large description de son architecture ainsi que des algorithmes d'optimisation qu'elle intègre [TGL98, TGL99b, TGL02, TGL99a, TLG97], nous en donnons ici un rapide résumé.

A la manière de la structure hiérarchique utilisée dans la bibliothèque *MPICH*, *ROMIO* exploite une couche d'abstraction lui permettant d'être proposée au dessus d'un grand nombre de systèmes de fichiers (cf. figure 3.3).

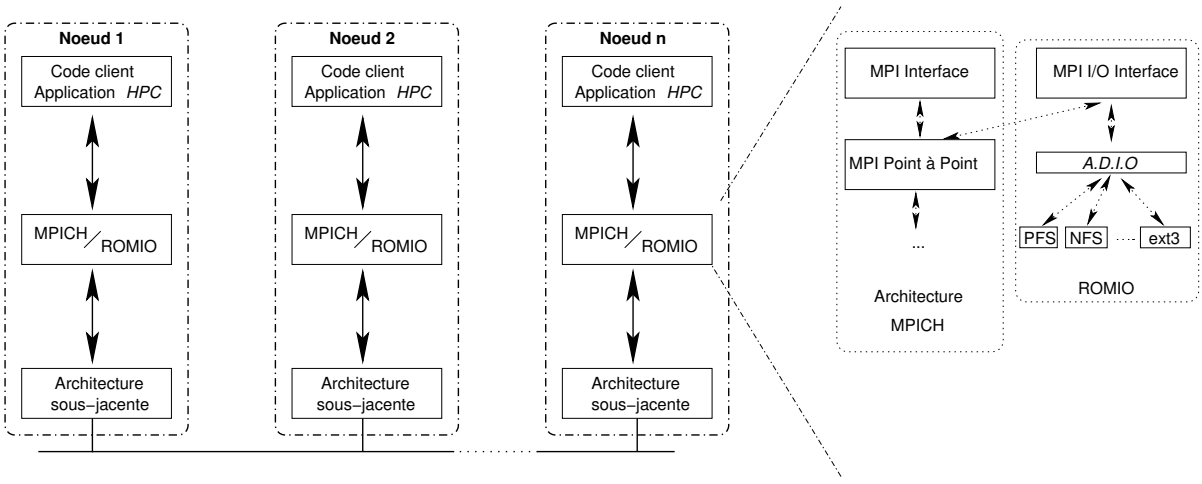


FIG. 3.3 – Architecture interne de *ROMIO*

[22] `http://www.mcs.anl.gov/romio.`

La couche «haute» fournit les routines MPI I/O, elle est commune aux différents sous-systèmes composant la couche basse du modèle assurant la portabilité (*ADIO* [23]).

Quatre niveaux d'accès sont définis au sein du standard :

- 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 participants de se synchroniser lors de chaque appel. L'intérêt de cette approche est de limiter une disparité trop importante dans les portions de fichiers auxquelles les processus accèdent. Dans le niveau 0, chaque processus doit effectuer 7 accès indépendants (représentés par les 7 traits). Le déploiement de l'application MPI par le lanceur mpirun engendre des décalages entre le premier processus et le dernier. Ainsi, le processus 0 peut avoir réalisé plusieurs requêtes et donc accèder au milieu du fichier alors que le dernier processus lui exécute seulement sa première requête. Ce comportement dégrade considérablement les performances, les mécanismes de pré-chargement n'étant pas exploités. Dans le niveau 1, la barrière de synchronisation utilisée à chaque appel permet aux processus d'«avancer» de manière parallèle, favorisant potentiellement l'utilisation du cache (même si l'opération est collective, les requêtes ne sont pas agrégées).
- Niveau 2 : *Data Sieving*, chaque processus définit son schéma d'accès (*datatype*) définissant les accès disjoints (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. Pour que cette technique soit efficace, il est impératif d'avoir un degré de continuité bien évalué (cf. annexe A). Par ailleurs, l'utilisation de cette technique est conseillée seulement lorsqu'un processus souhaite manipuler des données disjointes. En effet, la mise en place de cette approche sur l'ensemble des processus s'avère inutile en plus d'être extrêment coûteuse : chaque processus va recupérer la totalité des informations. Le système de stockage va devoir manipuler n fois la taille du fichier (n étant le nombre de processus participant à l'opération). Dans ce dernier cas, il est préférable soit d'effectuer l'opération depuis un processus et de collecter/redistribuer les données depuis/vers les processus concernés soit de faire appel à la stratégie collective, le niveau 3.
- Niveau 3 : *Two-Phase*, chaque processus définit le type correspondant à son schéma d'accès (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 (cf. 2.2.1.4). Il est important de noter que la totalité de l'opération d'écriture ou de lecture sur le fichier n'est pas réalisée par un seul processus mais répartie de manière équitable entre l'ensemble des processus, chacun d'entre eux accèdant à une portion contigüe du fichier.

Le schéma 3.4 extrait de [TGL02] illustre ces 4 niveaux.

Le standard propose la possibilité de partager un descripteur de fichier entre l'ensemble des processus d'une même application (cf. section 2.1.2).

Comme nous l'avons dit, le standard MPI I/O ne prévoit pas de routines permettant de gérer le placement physique des données sur le disque. En effet, le consortium MPI n'a pas

[23] *Abstract Device Interface for Parallel Input/Output*.

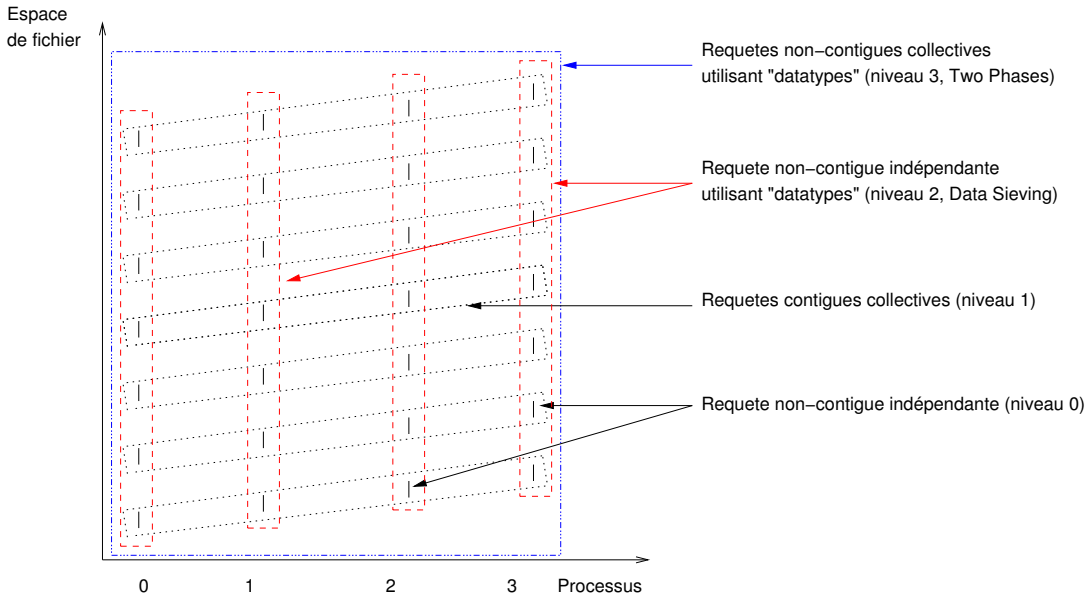


FIG. 3.4 – Niveaux d'optimisation définis dans le standard MPI I/O
Chaque rectangle représente la quantité de données manipulées pour chaque niveau d'optimisation.

souhaité intégrer ce type de service : la correspondance entre vue logique et vue physique est dépendante des unités de stockage employées, par conséquent, fournir des routines de placement diminuerait considérablement le coefficient de portabilité du modèle (et donc MPI). Dans un souci de souplesse, MPI I/O a prévu un ensemble de routines «génériques» permettant de transférer diverses informations depuis l'application vers les couches basses du modèle. Il est ainsi possible pour une application de transférer un ensemble de paramètres afin d'optimiser les E/S au niveau matériel. Toutefois, ces routines ne sont régies par aucune règle spécifique, ainsi rien n'impose leur présence au sein des couches basses. Les systèmes *PVFS* et *Lustre* qui proposent une interconnexion directe au niveau de l'*ADIO* proposent plusieurs paramètres permettant d'améliorer les performances de l'architecture déployée.

Mercutio

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

Le modèle se scinde en deux parties :

– La couche *BAFS* :
Elle est la couche basse de l'implémentation. Elle permet d'interfacer les différents systèmes de fichiers sous-jacents Tirant partie des concepts objets, la couche est relativement adaptable à tout nouveau type d'architecture. Comparable à l'*ADIO*, elle propose une interface d'entrées/sorties point à point efficace entre les différents nœuds et les divers systèmes de stockage. *BAFS* permet en autre d'abstraire plusieurs systèmes de fichiers au même moment (utilisation du concept du système de fichier virtuel) ou d'exploiter de front des unités de sauvegarde via des bibliothèques d'E/S à accès direct. L'asynchronisme totalement intégré au modèle et implanté au sein de ce sous-système : elle est mise

en œuvre soit par l'intermédiaire de routines non-bloquantes fournies par les systèmes de fichiers sous-jacents soit par l'exploitation de *threads* dans le cas contraire. Par ailleurs, l'agrégat des requêtes au sein d'un nœud (*Data Sieving*) est également effectué au sein du module *BAFS*. Ayant quantifié la notion de «degré de continuité» (cf. annexe A) durant la phase de conception, le modèle *MercutIO* tient compte de la distance qui sépare chaque bloc de données contigu ; si celle-ci est inférieure à la taille la plus petite des portions alors l'approche *Data Sieving* est mise en œuvre. Dans le cas contraire, l'approche standard *UNIX* est employée.

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

A ce jour, *MercutIO* a été incorporé au sein d'une implantation récente du standard `MPI 2`. Projet industriel à but commercial présenté en avril 2003, *ChaMPIon/Pro* [24] est disponible sous l'environnement *Red Hat Linux Enterprise*.

3.2.7. Bilan

Les bibliothèques ont une vue généralement moins fine du placement physique par rapport aux systèmes de fichiers parallèles qui intéragissent avec les couches basses de la pile des E/S aussi bien du côté client que du côté serveur. Cependant, elles ont une plus grande connaissance des schémas d'accès exploités par l'application parallèle et offrent des améliorations plus pertinentes. Un nombre important de solutions a été proposé depuis le milieu des années 90. Pour la plupart, elles ont été construites au-dessus d'un modèle à base de messages de type `MPI` ou `PVM`. Même si certaines ont proposé une interaction plus importante avec le système de fichiers sous-jacent, comme par exemple la solution *PASSION*, elles ont rapidement convergé vers les mêmes types de fonctionnalités : agrégation des accès disjoints et mise à disposition de routines asynchrones [25] .

Ainsi, dans un souci d'uniformisation des interfaces et reposant la plupart sur le modèle à base d'échanges de messages, le standard `MPI I/O` a été défini. Il spécifie un ensemble de routines standardisées (fondées sur 4 niveaux d'optimisation) pour la manipulation des données par une application parallélisée (cf. section 3.2.6.1). La bibliothèque *ROMIO* développée en suivant ce standard fait office aujourd'hui de référence.

Néanmoins comme nous allons l'indiquer dans un bilan général, en plus de nécessiter une prise en main de leur interface parfois délicate, ces solutions sont mono-applicatives ce qui est un réel inconvénient dans un environnement de type grappe où généralement plusieurs applications parallèles s'éxécutent de manière concurrente.

[24] `http://www.mpi-softtech.com`, une version parallèle de `MPI` (*MPI / Pro*) est également développée au sein du même organisme.

[25] Nous noterons tout de même que la bibliothèque *PPFS* offre en plus la possibilité de mettre un gestionnaire de cache applicatif.

Caractéristique \ Bibliothèque		*PASSION*	*PANDA*	*PPFS*	*Jovian*	*MTIO*	*Romio*	*Mercutio*
But		Problèmes *out-of-core*	Tableaux multi-dimensions	Gestion des E/S parallèles	Gestion des E/S parallèles	Gestion des E/S parallèles	Gestion des E/S parallèles	Gestion des E/S parallèles
Architecture		Répartie	Client/serveur	Client/serveur	Client/serveur	répartie	répartie	répartie
Fonctionnalités	**Placement des données**	??	OUI	OUI	OUI dans la version 2	NON	NON	NON
	Définition de vues	??	OUI	OUI	NON	??	OUI	OUI
	Accès recouvrant	OUI	OUI	OUI	OUI	OUI	OUI	OUI
	Accès collectif	OUI	OUI	Pas directement (*disk-directed I/O*)	version 1 : tout accès est collectif version 2 : aucun accès colllectif	OUI (prise en compte de la locatité des données)	OUI	OUI
	Accès asynchrone	??	OUI	??	NON	Par défaut (modèle asynchrone)	OUI	OUI
	Ordonnancement des accès	OUI	NON	NON	NON	NON	Pas directement	Pas directement
	Pré-chargement	OUI	OUI	OUI	NON	NON	NON	NON
	Cache	OUI	??	Cache hiérarchique paramétrable	NON	NON	NON	NON
Prérequis	**Système de fichiers**	*Intel CFS* ou *Vesta*	`UNIX`	`UNIX`	`UNIX`	`UNIX`	*UNIX*	`UNIX`
	Divers		`MPI`	`MPI`		`MPI`	`MPI`	`MPI`
Utilisation		Pas à notre connaissance	OUI (*RocPANDA*)	OUI (*PPFS II*)	Pas à notre connaissance	NON	OUI (référence actuelle)	OUI (version commerciale)

TAB. 3.2 – Récapitulatif des caractéristiques des bibliothèques présentées

3.3. Vers une solution `POSIX` multi-applicative

Tout au long de ce chapitre, nous avons présenté des solutions majeures proposées par la communauté scientifique des années 90 à aujourd'hui. Chacun de ces systèmes tente de réduire l'impact sur les performances engendré par les modes d'accès généralement employés dans les applications parallèles scientifiques : plusieurs accès disjoints transmis simultanément depuis plusieurs processus répartis potentiellement sur plusieurs nœuds et à destination d'un même fichier.

Comme nous allons le vérifier dans le prochain chapitre, cette dégradation est plus que significative. Elle est causée par le temps d'accès et au coût du déplacement des têtes de lecture/écriture (9 ms en moyenne) nécessaire pour servir la multitude d'accès disjoints émanant de l'application parallélisée.

Pour pallier ce problème, une première solution «matérielle» repose sur l'utilisation de plusieurs unités de stockage (*RAID* 0, cf. section 1.1.6) ou serveurs d'E/S afin de répartir l'ensemble des requêtes : plusieurs accès peuvent alors être gérés en parallèle et les débits sont globalement meilleurs. Malheureusement, cette approche a plusieurs limitations. Premièrement, chacune des unités n'est pas exploitée de manière fine : le débit «agrégé» (ou cumulé) de l'ensemble des unités permet de satisfaire en partie les besoins de l'application mais à un ratio coût/performance relativement important. Par ailleurs, le découpage utilisé pour répartir les données est prépondérant dans les performances : un accès d'un octet supérieur à la taille d'un bloc (*chunk*) va nécessiter deux appels. La répartition sera alors inefficace en plus d'engendrer un surcoût.

Des approches plus «logicielles» ont été proposées afin d'améliorer les performances, nous en avons présenté une synthèse lors du chapitre 2. Elles consistent à :

- intéragir avec les couches inférieures pour tenir compte/paramètrer le placement physique des données ;
- agréger les requêtes afin d'en réduire le nombre d'une part, et, d'autre part, d'accéder à des portions plus conséquentes servies plus rapidement par les systèmes de stockage ;
- exploiter les méthodes asynchrones pour tenter de recouvrir les phases de calcul et d'E/S ;
- mettre en place des structures de caches globales à l'application et/ou à l'architecture de stockage ;
- optimiser l'ordre d'émission des requêtes en utilisant des stratégies d'ordonnancement.

Les systèmes présentés dans ce chapitre combinent plusieurs de ces approches. Cependant, selon la catégorie à laquelle ils appartiennent, ils ont plus ou moins de facilité à proposer certaines fonctionnalités. La première catégorie, les systèmes de fichiers distribués (ou «parallèles» lorsqu'ils exploitent plusieurs serveurs d'E/S) permettent de prendre en compte l'architecture matérielle sous-jacente (capacité d'analyse, débits disques et réseaux, topologie de l'architecture). Ainsi, ils peuvent proposer les routines requises pour paramétrer un grand nombre de critères physiques comme la taille de répartition ou l'utilisation de vues physiques. Les systèmes de fichiers *PIOFS* ou *Galley* proposent de telles fonctionnalités. Cependant, la richesse de ces interfaces et la disparité entre plusieurs solutions n'a pas permis une éclosion de tels systèmes.

Des systèmes de fichiers «nouvelle-génération» moins spécifiques ont été suggérés à la communauté. Le système *PVFS*, ou encore le tout récent *Lustre*, en sont les actuels porte-flambeaux. Ils tentent de répondre à plusieurs critères très différents entre eux comme la cohérence des données, le passage à l'échelle ou encore la tolérance aux pannes et ne sont plus

dédiés aux E/S parallèles. Cependant, ils proposent généralement une interconnexion directe avec la bibliothèque *ROMIO* du standard `MPI I/O` et se «libèrent» ainsi de ces contraintes.

La solution applicative répondant au standard `MPI I/O` semble donc l'approche généralement conseillée. Cependant, la richesse des interfaces définie comme principal inconvénient des systèmes dédiés peut également être imputée au standard `MPI I/O` : l'interface comprend 58 routines dont plus d'une dizaine qui sont totalement dépendantes de l'architecture sous-jacente.

Ainsi, l'utilisation de bibliothèques mettant en œuvre les spécifications `MPI I/O` requiert deux points fondamentaux afin d'en tirer le meilleur gain :

- une profonde connaissance de la part des utilisateurs de l'ensemble des subtilités de chacune des fonctionnalités ;
- une étude minutieuse, tout d'abord, des caractéristiques et des performances du système sur lequel elles reposent (accès séquentiel vs parallèle, synchrone vs asynchrone, partagé vs dédié, granularité des unités de stockage, ...) mais également des couches intermédiaires qu'elles exploitent (`MPI`, `TCP/IP`, ...) permettant les communications inter-nœuds. Il est, en effet, impératif lors de la mise en œuvre de connaître toutes les éventuelles limitations pouvant être induites par ces sous modules (débits, passage à l'échelle, surcoût des opérations collectives ...).

Parallèlement, la problèmatique des E/S parallèles semble avoir été traitée dans tous les cas et sur tous les systèmes d'un point de vue mono-applicatif : une application composée d'un nombre plus ou moins important de processus qui accèdent de manière parallèle aux ressources de stockage. Certes la proposition de solutions pour améliorer les performances dans un tel cas est primordiale, cependant, et de notre point de vue, cette position n'est plus suffisante à l'heure actuelle. Les architectures de type grappe étant de taille de plus en plus conséquente et les gestionnaires de ressources de plus en plus aboutis, un nombre important d'applications sont exécutées de manière concurrente.

De ce fait, la direction choisie par les systèmes de fichiers nous parait étonnante. Laisser la gestion des E/S parallèles à des solutions applicatives nous semble inappropriée : chacune des applications ayant un impact sur les autres. Lorsqu'un programme utilise une bibliothèque d'E/S parallèles afin d'être plus efficace, l'ensemble des optimisations ne tient compte que des requêtes qui lui sont propres. De ce fait, les stratégies collectives généralement mises en œuvre pour favoriser des accès plus «séquentiels» s'avèrent inutiles et coûteuses lorsqu'elles entrent en conflit avec d'autres requêtes provenant d'applications concurrentes au niveau des serveurs de stockage.

Dans un contexte de calcul intensif et d'architecture parallèle distribuée, le système de stockage, local ou distant, semble le seul capable de prendre en compte la totalité des interactions émanant de l'ensemble des applications s'exécutant sur l'architecture parallèle et de mettre en œuvre les méthodes d'optimisation les plus adaptées. Une approche naïve consisterait à traiter les accès programme par programme : l'ensemble des accès serait récupéré au sein du système de fichiers, les requêtes seraient réordonnées de manière à favoriser les techniques de préchargement et les fichiers seraient manipulés l'un après l'autre. Cela fournirait d'excellentes performances globales mais aurait un impact significatif sur l'interactivité en plus d'engendrer des problèmes de famine, critères prépondérants dans une architecture multi-applicative comme l'est une grappe.

Malheureusement, la plupart des systèmes de fichiers actuels ne fournissent pas de politiques d'ordonnancement appropriées. Le système *PVFS* propose une stratégie adaptative (cf. section 2.2.4) mais qui ne tient compte que d'une application. Le système *Clusterfile System* utilise une heuristique mais pour maximiser l'utilisation des serveurs et ne se concentre pas sur les applications.

Les questions, qui nous ont guidé dans le travail décrit dans la seconde partie du document, ont été :

- **dans quelle mesure est il possible d'obtenir des performances en s'appuyant uniquement sur les routines `POSIX`**.

- Si cela est réalisable, **comment une politique d'optimisation globale à l'ensemble des applications s'exécutant sur la grappe peut elle être mise en œuvre**.

Dans le chapitre 4, nous présentons différentes évaluations concernant l'étude des E/S parallèles émanant d'une application au-dessus d'une grappe de l'architecture *Grid 5000*. Les expériences se sont déroulées tout d'abord dans un contexte mono puis multi-applicatif. Cette étude nous a permis d'étudier l'impact des applications et de déterminer un élément fondamental à notre proposition à savoir la sérialisation.

II

La proposition *alOLi*

However beautiful the strategy,
you should occasionally look at the results !
Winston Churchill

ETUDE PRÉLIMINAIRE 4

Comme nous l'avons décrit dans les chapitres précédents, les modes d'accès parallèles mis en œuvre dans les applications scientifiques modernes affectent considérablement l'efficacité des systèmes de stockage sous-jacents.

Afin de limiter l'impact et fournir de meilleures performances, des solutions basées sur du pré-chargement en mémoire volatile ou sur des interfaces d'accès plus spécifiques sont souvent appliquées (cf. chapitre 2 et 3). La mise en œuvre de ces approches a pour but d'obtenir une mode d'accès séquentiel à l'opposé d'un comportement «pseudo-aléatoire» moins performant (mauvaise utilisation du cache, repositionnement des têtes, ...).

Dans ce chapitre, nous évaluons de tels comportements sur un serveur centralisé NFS [1] . Les diverses expérimentations décrites valident les remarques établies dans les chapitres précédents. Par ailleurs, elles permettent d'introduire un des concepts fondamentaux de notre proposition, à savoir la sérialisation des accès. Cette notion, comme nous le verrons, permet de reproduire, pendant l'exécution, un comportement séquentiel sans pour autant requérir à une interface spécifique et sans à avoir à utiliser des mécanismes de synchronisation globale souvent requis dans les solutions actuelles.

[1] Même si le système NFS n'est pas le plus adéquat pour les architectures parallèles dédiées au calcul intensif, il reste le standard pour les configurations de petite et moyenne taille (cf. section 3.1.1).

4.1. Plate-forme et plan d'expérience

Nous présentons, dans cette section, la plate-forme utilisée pour nos différentes expérimentations ainsi que les applications permettant de reproduire les modes d'accès présents sur les grappes. Les paramètres de configuration ainsi que plusieurs justifications pour le déroulement des expérimentations y sont donnés.

4.1.1. Plate-forme

Les expériences et mesures effectuées ont été réalisées, sauf mention spéciale, sur la grappe de Sophia-Antipolis de l'architecture GRID5000. La grappe est composée d'une centaine de noeuds bi-processeur (AMD Opteron cadencé à 2GHz, 1 Mo de cache et 2 Go RAM). Chaque machine est connectée via un interface Gigabit Ethernet. Les unités de stockage présentes sur les machines sont des disques IDE de 80 Go (Maxtor 6Y080L0 ATA 133, 2 Mo de cache, temps de positionnement moyen annoncé : 9 ms) avec un débit, mesuré par la commande `hdparm -t`, d'environ 57 Mo/s.

Le système d'exploitation *GNU/Linux* a été déployé pour chaque expérience avec un noyau 2.6.12 puis 2.6.15 pour les dernières expériences. Néanmoins, même si le protocole `NFS` version 4 apparaît dans les dernières versions, seul le protocole `NFS` V3 a été utilisé et les performances n'ont, par conséquent, pas subi de modification entre ces deux versions.

Pour chaque expérience, un nœud dédié jouait le rôle de serveur `NFS` en exportant, seulement aux machines souhaitées, une partition `ext3` [2] . La granularité d'accès retenue pour `NFS` a été fixée à 32 Ko pour les lectures et les écritures. Les caches `NFS` côté client sont désactivés (les valeurs `actimeo` et `noac` utilisées pour monter la partition `NFS` sont assignées à zéro). Le protocole `TCP` est utilisé afin de limiter la retransmission des requêtes RPC non traitées (`timeout` à 7 sec et à 60 secondes pour `TCP`, `TCP` assurant lui-même le contrôle). A chaque mise en place du serveur `NFS`, la partition exportée a été reformattée. Les partitions ont été démontées entre chaque expérience afin d'éviter tout effet de cache. Enfin, les fichiers «tests» ont été créés via les commandes `dd` toujours dans le même ordre afin d'éviter les problèmes de placement (cf. effet *ZCAV*, section 1.1.5). Chaque expérience manipule au moins 4 Go (2 fois la taille de la mémoire). La plupart du temps, le facteur mesuré correspond à la bande passante attribuée à chacune des applications. Dans certains cas cependant, nous utilisons les temps de complétion (ou temps de terminaison) nous permettant d'avoir une meilleure vision des différences de performance. En effet, nous souhaitons aborder simultanément le critère de performance globale mais également celui de l'équité entre les applications ; un débit global ou un temps de complétion correspondant à la dernière application ne nous permettrait pas d'analyser ce dernier critère.

4.1.2. Décomposition de fichiers

Les applications scientifiques parallèles s'appuient sur des modes d'accès particuliers. Par exemple, lors d'un produit parallélisé de matrices, chaque processus doit récupérer des parties spécifiques des deux matrices en fonction du découpage prédéfini (colonne, ligne, BLOCK, CY-

[2] Le système de fichiers `ext3` est le système de fichiers local généralement mis en œuvre au sein des systèmes *GNU/Linux*.

CLIC, BLOCK/BLOCK, ...). C'est ce type d'opération que nous désignons par «décomposition de fichiers».

Le but de cette première étude consiste à évaluer les performances engendrées par des opérations de ce genre. Pour cela, il nous est nécessaire de reproduire les techniques généralement employées pour réaliser ces décompositions.

Deux approches sont généralement employées :

- La première consiste en deux étapes : un client, choisi arbitrairement, rapatrie les données en local de manière séquentielle afin de maximiser l'efficacité. Lors d'une seconde phase, il redistribue selon le découpage prédéfini, les données aux nœuds concernés. Cette technique, similaire à la solution de *prefechting*, requiert d'une part un réseau de communication performant en vue de limiter les surcoûts imposés par le transfert redondant des données et d'autre part une quantité de mémoire significative afin d'éviter les problèmes de type *out-of-core*. Cependant, elle est, quand cela est possible, préférée car plus simple.

  Les opérations de ce type ont été réalisées par une application «`cat`-like» reproduisant un comportement séquentiel synchrone : le fichier est lu ou écrit du début à la fin. Une nouvelle requête est transmise à la réception de la précédente.
- La seconde méthode consiste à récupérer directement et à partir de chaque processus les données désirées. Dans ce cas, le serveur distant reçoit en parallèle plusieurs requêtes à différents *offsets* et de différentes tailles. Comme nous l'avons vu, ce type de comportement affecte considérablement les performances et génère des phénomènes de congestion au niveau du système de fichiers. L'utilisation d'interfaces à la `MPI I/O` permet d'obtenir généralement de meilleures performances.

  Les décompositions avec mode d'accès parallèle ont été évaluées en utilisant le jeu de tests «`IOR` [3] » proposé par le *Lawrence Livermore National Laboratory*.
  Ce programme permet d'évaluer les performances d'un système de stockage lors d'accès en parallèle basés soit sur les interfaces `POSIX`, soit sur les interfaces `MPI I/O`. Le découpage généré, correspond à une répartition cyclique entre l'ensemble des processus participants. La granularité du découpage est donnée au lancement du programme. Afin de pouvoir comparer les différentes approches avec les interfaces `MPI I/O`, nous avons fait varier la granularité de 8 Ko à 4 Mo (4 Mo étant la taille maximale des tampons internes de la bibliothèque `ROMIO`).

Le nombre de requêtes émanant d'une décomposition de 4 Go pour chaque instance (en fonction de la granularité de découpage) est illustré à la figure 4.1.
Les courbes 1, 4, 8, 32 correspondent au nombre de processus (ou d'instance `MPI` dans notre cas) participant à la décomposition. Ce graphique permet d'introduire plusieurs aspects. Tout d'abord, il permet de voir le nombre de requêtes que génère chacune des instances : plus le nombre de participants est important, plus le nombre de requêtes par processus est petit mais plus le serveur reçoit des demandes en parallèle (et inversement). La courbe «Protocole `NFS`» correspond au sous-découpage imposé par le protocole `NFS`. Comme nous l'avons indiqué, la granularité retenue a été fixée à 32 Ko : ainsi lorsqu'un processus dépose, par exemple, une requête de 4 Mo, celle-ci est scindée en plusieurs requêtes de 32 Ko. De ce fait, à partir de granularité de découpage dans `IOR` supérieure à 32 Ko, le nombre de requêtes `NFS` délivré par

[3] *Interleaved Or Random*, le nom vient d'une ancienne version qui proposait, en plus d'un mode interlacé, un mode d'accès aléatoire, `http://www.llnl.gov/asci/purple/benchmarks/limited/ior/`.

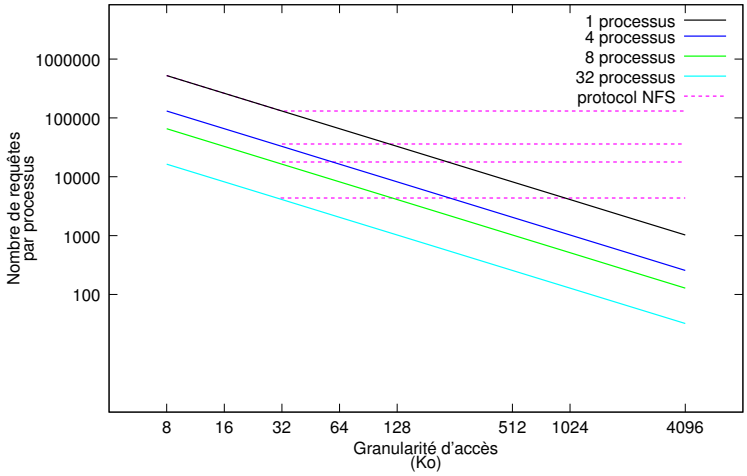


FIG. 4.1 – Distribution des requêtes par instance pour une décomposition de 4 Go

chaque processus reste le même (affectant d'autant plus les performances). Nous reviendrons par la suite sur cette dépendance vis à vis de la granularité sous-jacente.

4.1.3. Calculateurs parallèles et implantation de `NFS`

Une caractéristique fondamentale des super-calculateurs est que quelque soit le nombre de processeurs qui les composent, un unique noyau contrôle l'ensemble des évènements liés aux E/S fichiers.

Dans l'approche classique du traitement d'une requête (une requête `read` ou `write`), le client «transmet» sa demande à la pile d'E/S et est mis en attente jusqu'à réception. L'implantation des couches clientes `NFS` selon les systèmes d'exploitation «`UNIX`-*like*» diverge [Bou02].

Deux implantations sont principalement mises en œuvre :

- Dans la première, plusieurs *threads*, `biod` (*block I/O dameon*), sont en attente de requête côté client. Lorsqu'une requête est transmise par un `biod` au serveur `NFS`, le *thread* `biod` bascule dans un état bloqué jusqu'au retour de la réponse. Son comportement est donc synchrone et le nombre de `biod` définit le nombre de requêtes pouvant être gérées en parallèle. Ainsi, lorsque plusieurs processus, qui s'exécutent sur un même nœud, souhaitent accéder à des données distantes, ils se retrouvent en concurrence. De ce fait, un processus qui dépose plusieurs requêtes et qui, par conséquent bloque l'ensemble des démons `biod`, monopolise le service au dépend des autres processus.
- La seconde solution, implantée dans les systèmes *GNU/Linux*, repose sur un seul *thread* noyau, nommé `rpciod`. Ce *thread* transmet les requêtes de manière asynchrone au serveur. Chaque demande est déposée dans une queue traitée par le démon `rpciod`. Lorsqu'une requête a été transmise au serveur, le démon démarre le traitement de la requête suivante, permettant ainsi de recouvrir les temps d'attente et réduisant ainsi le déséquilibre entre les processus. Néanmoins, lorsque le nombre de requêtes est important pour chaque processus (par exemple, une décomposition par granularité de 4 Mo engendre à chaque appel 128 requêtes `NFS` de 32 Ko), l'inégalité peut survenir à nouveau.

Par conséquent, afin d'éviter tout phénomène de déséquilibre ou encore de congestion côté client, chaque instance a été déployée sur un nœud de manière isolée. Il est ainsi, le seul à accéder à la couche client du service `NFS`.

4.2. Courbes de référence

Le but de cette section est de présenter les différentes valeurs qui nous serviront de «référence» tout au long de nos expérimentations.

Depuis la version 3, le système `NFS` présente un mode `async` permettant d'améliorer les performances en écriture : chacun des clients peut transmettre plusieurs requêtes d'écriture sans pour autant avoir reçu les acquittements de chacune d'entre-elles (cf. section 3.1.1). En effet, il est important de distinguer les accès en lecture (sollicités) des requêtes d'écriture (potentiellement sollicitées). Une requête de lecture entraîne une action sur le disque lors de son premier chargement en mémoire. Au contraire, un requête d'écriture peut être simplement déposée dans un tampon temporaire et être traitée par la suite. Ainsi, la politique de cache choisie peut avoir un impact considérable sur les performances mais également sur l'intégrité des données en cas de panne (des données pouvant par exemple être perdues en mode asynchrone, cf. 1.1.4).

Nous avons évalué les différents possibilités de configuration de montage pour le client `NFS` ainsi que la manière dont est montée la partition exportée par le serveur. Le but a été de déterminer les performances susceptibles d'être atteintes en fonction du risque potentiel de perte de données. Le tableau 4.1 décrit les configurations évaluées.

	client	serveur	Intégrité en cas de défaillance
Cas 1	`mount sync`	`sync`	+ +
Cas 2	`mount async`	`sync`	+
Cas 3	`mount sync`	`async`	-
Cas 4	`mount async`	`async`	- -

TAB. 4.1 – Paramétrage du service `NFS`

Les résultats obtenus lors de la récupération d'un fichier de 4 Go de manière séquentielle en fonction d'une granularité variant de 8 Ko à 4 Mo apparaissent sur la figure 4.2. La courbe en pointillé correspond au débit maximum délivré par l'unité de stockage, à savoir 57 Mo/s.

Pour les lectures, les paramètres de synchronisation n'ont pas d'impact significatif et les performances sont toutes comparables (autour de 46 Mo/s). Lorsque la granularité augmente, une légère diminution des performances apparaît toutefois. Elles est due à la granularité du protocole `NFS` : la requête est divisée en sous requêtes de 32 Ko puis transmise de manière «parallèle» au serveur. De ce fait, l'ordre des requêtes, au niveau du serveur, n'est pas forcément garanti parmi ce sous-groupe ; le mode est comparable à un comportement parallèle à petite échelle.

L'évaluation des accès en écriture dépend, en plus des paramètres utilisés pour accéder aux données, de la manière dont l'opération d'écriture a été implantée au sein d'un programme *HPC*. Deux approches sont généralement mises en œuvre : la première consiste à écrire les données sans se préoccuper de la gestion des méta-informations alors que la seconde pré-alloue la taille totale du fichier avant de démarrer l'opération. Cette étape supplémentaire permet généralement aux couches du système de fichiers sous-jacent d'essayer d'optimiser le placement des blocs et donc de maximiser la contiguïté. Les deux modes ont été évalués. Les graphiques apparaissant au sein de la figure 4.3 présentent les valeurs mesurées.

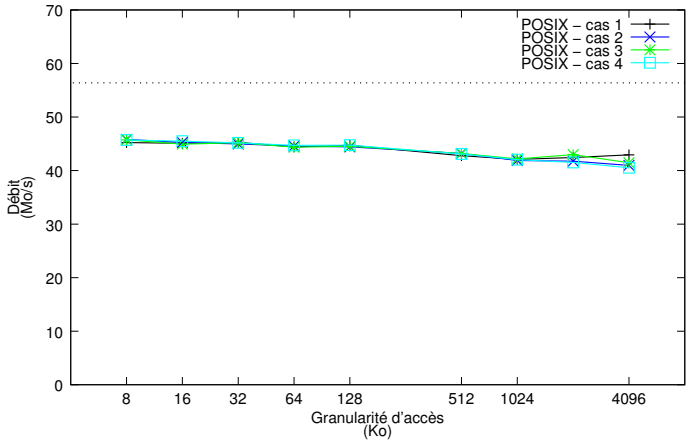
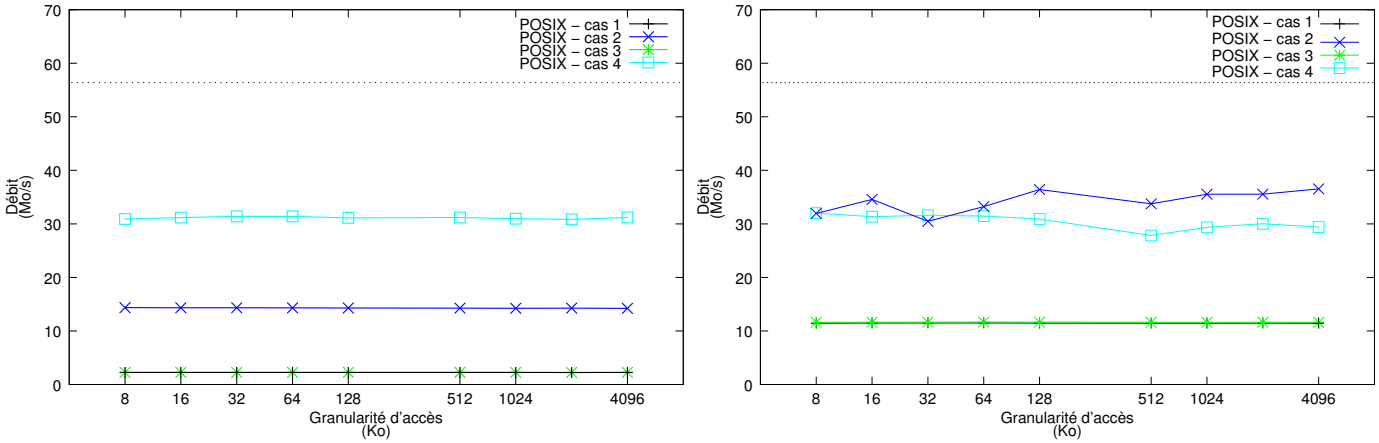


FIG. 4.2 – Lecture séquentielle d'un fichier de 4 Go
Les différents paramètres du protocole `NFS` sont évalués afin de déterminer la configuration la plus performante.



Sans pré-allocation Avec pré-allocation

FIG. 4.3 – Écriture séquentielle d'un fichier de 4 Go
Les différents paramètres du protocole `NFS` sont évalués afin de déterminer la configuration la plus performante.

Pour les écritures, le paramètre de synchronisation côté client semble primordial pour les performances. Les cas 1 et 3 ne fournissent pas de bons résultats pour les deux modes. Cependant, la pré-allocation semble bénéfique lorsque la partition exportée par le serveur est «montée» en mode synchrone. Le cas 1, qui assure l'intégrité des données, fournit des performances quasi-similaires au cas 2 (graphique de gauche) lorsqu'il est associé à la pré-allocation. D'un point de vue performance l'option `async` côté client semble obligatoire. Les courbes retenues comme «référence» sont le cas 4 et le cas 2 selon que les données aient été écrites respectivement sans ou avec la pré-allocation. A ce jour, nous n'avons pas réussi à expliquer pourquoi lorsque le fichier est pré-alloué, les performances sont légèrement meilleures lorsque la partition sur le serveur est «montée» en mode synchrone [4] .

[4]Plusieurs aspects n'ont pas été clairement identifiés dans la propagation des accès en mode synchrone pour le serveur `NFS` (`man 2 open, RESTRICTIONS`)

D'un point de vue plus général, les performances en lecture bénéficient des stratégies de *read ahead* présentes sur le serveur et sont par conséquent meilleures que celles fournies pour les écritures.

Les valeurs obtenues grâce à cette première expérience vont nous servir de références par la suite. Elles correspondent en effet au temps nécessaire pour récupérer ou écrire le fichier lorsque la décomposition est gérée de manière séquentielle : un seul processus sur un unique nœud récupère/écrit l'ensemble du fichier. Le temps nécessaire à la redistribution (en cas de lecture) ou la récupération des données (en cas d'écriture) serait, certes, à ajouter. Cependant, nous ne tenons pas compte de ce surcoût puisque le but n'est pas de comparer laquelle des deux approches est la meilleure mais de bien se concentrer sur les performances maximales qu'il est possible d'atteindre. Un des objectifs de ces travaux est de se rapprocher de ces performances séquentielles mais en accédant aux données directement depuis chacun des nœuds.

4.3. Évaluation des performances avec une seule application

Dans la suite de ce chapitre, nous allons nous attarder principalement sur une opération de décomposition de 4 Go en lecture et les divers conséquences qu'elle engendre. Nous reviendrons sur les accès de type écriture dans le chapitre 7.

4.3.1. `POSIX` et pile d'E/S standard

Le jeu de tests `IOR` paramétré pour employer l'interface `POSIX` est utilisé. L'opération est exécutée sur 4, 8 puis 32 processus répartis respectivement sur 4, 8, et 32 nœuds. Le client et le serveur `NFS` sont configurés selon le cas 2 défini au paragraphe précédent. Le graphique 4.4 présente les performances obtenues pour une granularité de découpage de 8 Ko à 4 Mo.

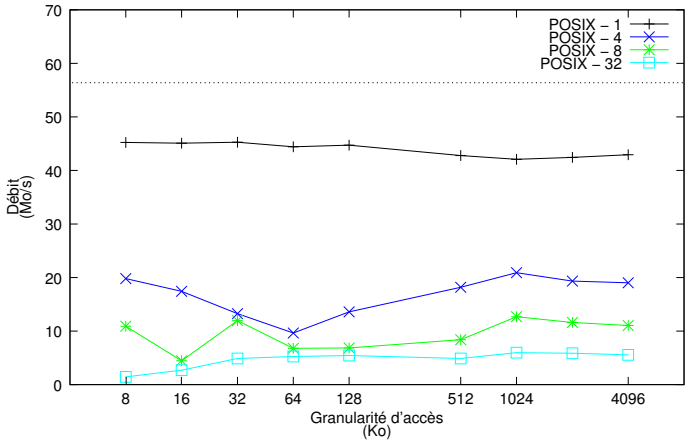


FIG. 4.4 – Décomposition d'un fichier de 4 Go (`IOR`)
La première courbe sert de référence. Les courbes 4, 8 et 32 correspondent aux diverses exécutions avec 4, 8 et 32 instances `MPI`.

Les performances sont intimement liées au nombre de clients participant à la décomposition : plus il y a de processus qui participent à la décomposition plus la dégradation est visible.

A titre indicatif, le fait que le débit soit légèrement meilleur pour les courbes présentant des décompositions parallèles (4, 8 et 32), pour les grosses granularités, est lié au décalage entre les

instances de MPI [5] . De ce fait, le premier processus est seul pendant un moment et accède de manière pseudo-séquentielle à ses différents blocs de 4 Mo. Réciproquement, le dernier processus bénéficiera à la fin de l'application du même phénomène. Il nous a paru intéressant de noter ce point puisqu'à l'opposé, le comportement purement séquentiel «POSIX - 1» lui subit une légère dégradation dans ce même cas. Nous reviendrons par la suite sur ce point (cf. section 4.3.3).

Suite à cette évaluation, nous avons voulu comprendre pourquoi les ordonnanceurs bas niveau ne jouaient pas leur rôle en réordonnant les requêtes du côté du serveur de stockage. Nous aborderons ce point dans la section suivante.

4.3.2. Évaluation des ordonnanceurs bas niveau

La mise en œuvre d'algorithmes d'ordonnancement d'E/S au niveau le plus bas du noyau (cf. section 1.2.2.3) a pour but de satisfaire un critère d'optimisation précis. Par exemple, l'algorithme *anticipatory* (stratégie activée par défaut dans les environnements *Linux*) est employé afin de maximiser le débit du disque tout en évitant les problèmes de famine. Lorsque plusieurs processus accèdent en parallèle à un même fichier, elle permet de réordonner les requêtes afin de fournir le mode d'accès le plus efficace.

Toutefois, comme nous avons pu le constater précèdemment, les performances diminuent considérablement lorsqu'une application accède à un fichier distant de manière parallèle. La première explication vient du fait que par défaut 8 démons nfsd sont présents dans l'espace noyau du serveur. Chaque démon ayant un comportement synchrone (lorsqu'il reçoit une requête, il la dépose dans la pile des E/S côté serveur et bascule dans un état bloqué jusqu'à la complétion de celle-ci), le nombre maximal de requêtes dans la queue de l'ordonnanceur bas niveau correspond au nombre de démons nfsd (8 requêtes dans le cas précédent).

Nous avons donc choisi de ré-exécuter l'expérience sur chacun des ordonnanceurs bas niveau en faisant varier le nombre de démon nfsd présent sur le serveur de 8 à 512 (la décomposition étant réalisée sur 32 processus répartis sur 32 nœuds).Les différentes courbes obtenues sont présentées sur la figure 4.5.

Indépendamment de l'ordonnanceur bas niveau et du nombre de *thread* nfsd déployé, nous pouvons constater que les performances n'atteignent globalement pas celles de référence (46 Mo/s de moyenne quelque soit la granularité, cf. figure 4.2).

En analysant les résultats obtenus par les deux premiers algorithmes (*anticipatory* et *deadline*), nous pouvons observer qu'ils fournissent des bonnes valeurs pour des granularités de 128 Ko avec un nombre de démons nfsd supérieur à 128. L'algorithme *deadline* permet même d'obtenir des valeurs supérieures, proches du débit maximal fourni par le disque (55 Mo/s). A cette granularité, chaque instance émet une requête qui est divisée en quatre sous-accès de 32 Ko. Puisque 32 processus participent à la décomposition, le nombre de requêtes pour un instant donnée (128) correspond au nombre de *threads* nfsd présents (il y a assez de *thread* pour traiter chaque requête). Il serait alors possible de présumer qu'il est nécessaire d'avoir autant de *threads* nfsd que de requêtes afin de pouvoir rétablir l'ordre séquentiel. Malheureusement, le raisonnement présenté ici n'est pas applicable à d'autres granularités qui nécessiteraient moins

[5] Lors du déploiement de l'application IOR par le lanceur mpirun, des décalages apparaissent entre le premier processus et le dernier. Aucune barrière de synchronisations n'est utilisée dans le cadre de l'utilisation de l'interface POSIX.

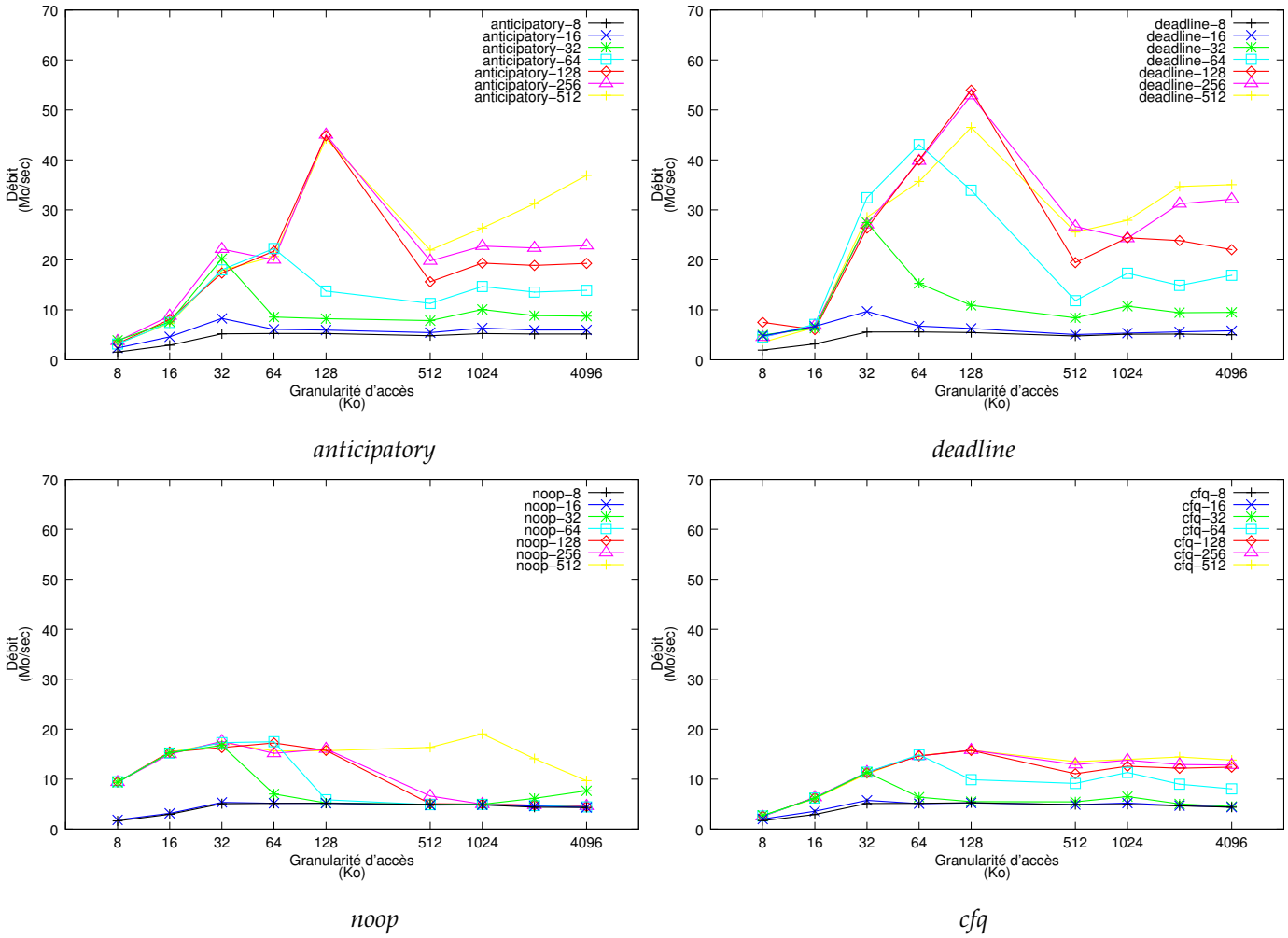


FIG. 4.5 – Impact du nombre de *threads* `nfsd` en fonction de l'ordonnanceur bas niveau utilisé
Décomposition d'un fichier de 4 Go (`IOR`) par 32 instances `MPI` réparties sur 32 nœuds.

de démons. Le temps d'arrivée des requêtes et les décalages entre les différents processus affectent considérablement les chances d'avoir les bons accès, au bon moment, dans la queue de l'ordonnanceur. Même le délai d'anticipation exploité dans la première stratégie, ne permet pas de rétablir l'ordre efficace.

L'algorithme *noop* n'apporte aucune optimisation, il transmet les requêtes aux disques sans aucun changement. L'évaluation de cet algorithme nous permet cependant d'observer l'interaction entre le noyau *Linux* et les stratégies d'ordonnancement du disque. Plus le nombre de démons est important, plus longtemps le débit fourni est maintenu autour de 17 Mo/s. Enfin et comme prévu, l'algorithme *cfq* n'apporte également aucune optimisation d'un point de vue performance ; son but, nous le rappelons, consiste à équilibrer les accès entre les applications.

Nous venons de voir qu'indépendamment du nombre de *threads* `nfsd` présents sur le serveur, les stratégies d'ordonnancement sont dépendantes de l'implantation du serveur et de la manière dont sont déposées les requêtes.

Dans l'étape suivante, nous avons voulu déterminer l'impact des comportements parallèles sur les stratégies de *read ahead* et sur le nombre de déplacements engendré au niveau du disque. Pour cela, nous avons évalué la récupération d'un fichier de 4 Go de manière synchrone (`cat`-like) mais dans un ordre aléatoire.

4.3.3. Sérialisation *vs* comportement parallèle

Nous avons constaté que les comportements d'E/S parallèles affectent de manière significative les performances pour une unité de stockage. Dans cette section, nous souhaitons savoir si une approche «sérialisée» permettrait de bénéficier en partie des techniques de type *read-ahead* en plus de diminuer le nombre de repositionnements au niveau du disque. Un comportement synchrone ou «sérialisé» se différence d'un mode séquentiel par le fait que l'offset de chaque nouvelle requête n'est pas forcément supérieur au précédent (cf. section 2.1.4). Même si le comportement séquentiel (et contigu) pour une unité fournit les meilleures performances, une approche sérialisée devrait, de notre point de vue, avoir un impact moindre sur le système de stockage et donc être plus efficace.

L'idée repose sur le fait qu'un accès sérialisé, dans un ordre non déterminé, entraîne moins de déplacements pour récupérer la totalité des données. En effet, si aucun ordre n'est établi lors d'un comportement parallèle, il y a, en plus des déplacements potentiels entre chaque bloc, des déplacements «internes» entre chaque accès. De ce fait, la sérialisation a plus de chances de bénéficier des stratégies de pré-chargement. Le schéma 4.6 illustre ces propos. Le gain apporté par une bonne utilisation des caches ainsi qu'une diminution des repositionnements devrait permettre de recouvrir la perte de temps liée à la sérialisation de deux accès (le disque n'étant pas exploité entre l'émission de la réponse depuis l'unité de stockage et l'envoi de la requête suivante par le client).

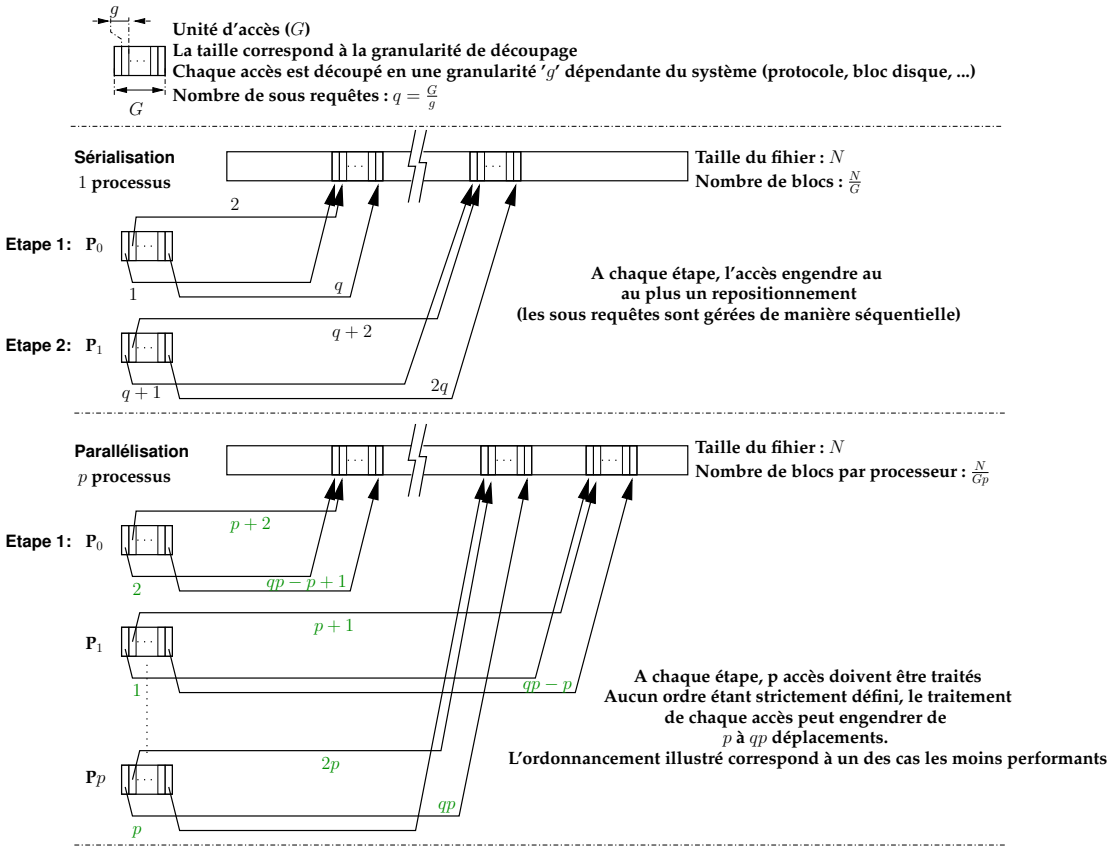


FIG. 4.6 – Comparaison des déplacements entre un comportement parallèle et sérialisé
Le comportement sérialisé engendre moins de déplacements et profite des stratégies *read-ahead*.

Plus la granularité d'accès au niveau de l'application est importante, plus la sérialisation doit se révéler efficace.

Par exemple, pour un fichier de 4 Go, plus de 500000 déplacements sont possibles pour une granularité de 8 Ko et seulement 1000 avec une granularité de 4 Mo. L'accès de 4 Mo correspondant à plusieurs lectures «blocs disques» devrait profiter des techniques *read ahead*. Lors d'un comportement parallèle, le même nombre de déplacements est requis d'un point de vue global (1000 déplacements répartis sur les 32 processus). Toutefois, chacune des requêtes de 4 Mo étant déposée de manière parallèle dans la file des requêtes disques, des déplacements entre ces différents sous-accès peuvent être engendrés et donc rendre inefficaces les techniques de pré-chargement.

La remarque réalisée lors de la section 4.3.1 concernant la légère augmentation du débit pour les grosses granularités due au décalage des instances, nous a amené à étudier ces aspects.

Afin d'oberver le gain susceptible d'être obtenu par une approche sérialisée, nous avons évalué la récupération d'un fichier de 4 Go de manière désordonnée depuis un processus.

La figure 4.7 reprend les courbes précédentes et montre les performances atteintes (courbe «`POSIX - 1 random`»).

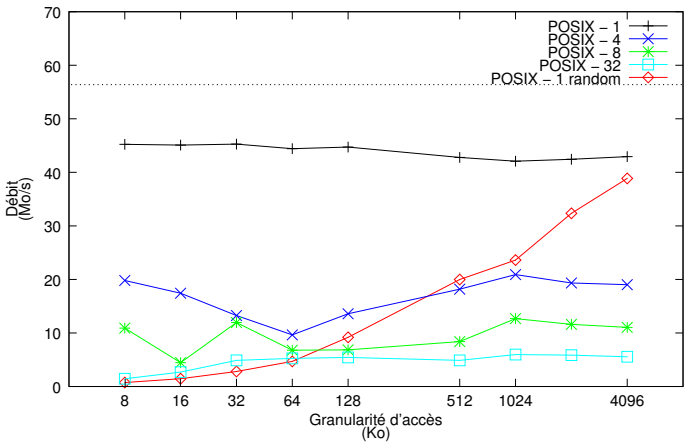


FIG. 4.7 – Décomposition d'un fichier de 4 Go (`IOR`)

La courbe révèle que selon le nombre de processus participant à la décomposition, il existe une granularité à partir de laquelle il est plus performant d'accéder au fichier de manière sérialisée, et ce même dans le désordre, que de récupérer le fichier par un mode d'accès parallèle (plus le nombre de participants est élevé, plus ce point apparaît avec une granularité petite : autour de 512 Ko pour 4 et autour de 128 Ko pour 8 et 32 instances). Par conséquent, transmettre les requêtes une à une à partir d'un certain seuil, tend à réduire ce phénomène.

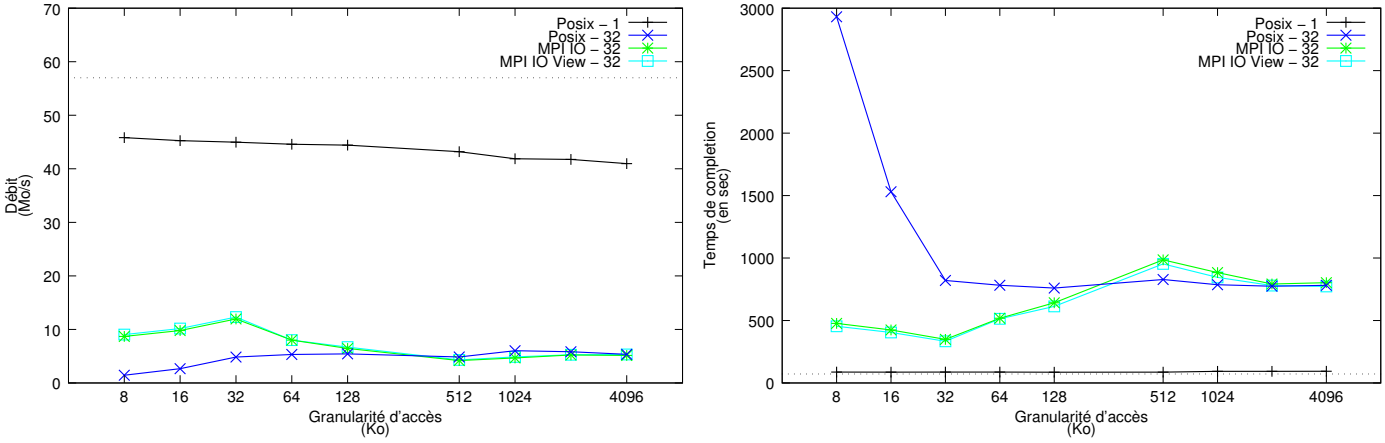
Avant de conclure cette première série d'évaluations sur une seule application (étude mono-applicative), nous allons présenter les valeurs références obtenues avec l'interface `MPI I/O` pour une décomposition réalisée par 32 clients.

4.3.4. Apports et lacunes des routines `MPI I/O`

La dernière expérience que nous avons réalisée dans le cadre mono-applicatif a consisté à évaluer le comportement du jeu de tests `IOR` paramétré pour utiliser l'interface `MPI I/O`.

Plusieurs configurations sont possibles (opération collective, pointeur de fichier partagé, définition de vues ...), toutefois seuls les niveaux 2 «opération collective» et 4 «opération collective associée à l'utilisation des vues» (cf. section 3.2.6.1), ont été évalués.

La figure 4.8 permet de comparer les valeurs atteintes par rapport à l'interface standard `POSIX`. Le graphe de gauche présente les résultats en fonction du débit alors que celui de droite donne le temps de complétion. Les courbes «`POSIX - 1`» correspondent toujours à la référence produite en section 4.2.



Débit Temps de complétion

FIG. 4.8 – Décomposition d'un fichier de 4 Go (`IOR`) sur 32 processus
Comparaison des interfaces `POSIX` vs `MPI I/O`.

La première remarque concerne le recouvrement des courbes «`MPI I/O`» et «`MPI I/O View`» qui nous a paru surprenant : les deux approches mettant en œuvre des concepts différents (cf. section 3.2.6.1 niveau 1 et niveau 3). Après avoir contacté la personne en charge du développement du jeu de tests au *LLNL*, il s'avère que le jeu `IOR` ne permet pas de définir des vues sur un espace supérieur à 2 Go et que le programme n'est, pour le moment, pas capable de combiner deux segments de 2 Go. Nous reviendrons sur ce point durant le bilan réalisé au chapitre 8. L'ensemble des comparaisons `POSIX/MPI I/O` avec le jeu de tests `IOR` sera, par conséquent, uniquement réalisé sur l'interface collective «simple» proposée par la bibliothèque `ROMIO`. Par ailleurs, l'utilisation de cette interface reste relativement accessible et même si elle fait appel à des routines spécifiques, celles-ci ne complexifient pas réellement le code. La comparaison avec l'interface `POSIX` n'en est que plus justifiable.

D'un point de vue général, l'approche collective proposée par `ROMIO` permet d'améliorer légèrement les performances pour les granularités allant jusqu'à 32 Ko. Une synchronisation est réalisée au sein de la bibliothèque avant chaque appel. Ainsi, l'ensemble des processus progresse de manière similaire et le décalage généré au démarrage de l'application est corrigé. Entre chaque barrière de synchronisation, le serveur reçoit 32 requêtes réparties sur un espace de $32 * taille\_granularite\_acces$. Cet espace borné limite les possibilités de déplacements en plus de favoriser potentiellement les stratégies de *read-ahead* lorsque la granularité est petite. Pour des tailles d'accès plus importantes, les requêtes étant subdivisées en sous accès de 32 Ko, le problème lié au comportement parallèle réapparaît et les performances se dégradent pour rejoindre celles produites par l'approche `POSIX`. Par ailleurs, cette barrière de synchronisation

qui semble favorable pour les faibles granularités, constitue un frein aux performances globales en sous-exploitant l'unité de stockage pendant les temps d'attente.

Nous n'avons pas remis la courbe correspondant au comportement sérialisé afin de ne pas surcharger les figures. Toutefois, en comparant avec la courbe «POSIX - 1 random» de la figure 4.7, nous pouvons constater que pour des granularités conséquentes, cette approche est plus efficace que la stratégie collective appliquée par la bibliothèque ROMIO. En effet, pour que les performances ne se dégradent pas lors de l'utilisation de l'interface MPI I/O, il est impératif de faire appel à des routines supplémentaires («*hints*», cf. section 3.2.6). Ces fonctions, propres à chacun des systèmes de fichiers, permettent de définir le comportement adéquat (ici par exemple, sérialiser les accès synchronisés) mais affecte la portabilité, critère prépondérant pour ce type de solutions (le code devenant dépendant d'une architecture de stockage particulière).

Le graphe de droite révèle l'impact d'un comportement d'E/S parallèles sur les performances. Le temps nécessaire pour terminer la décomposition en parallèle, «POSIX - 32» requiert plus de 45 minutes alors que le temps local à l'unité de stockage pour fournir une telle quantité (courbe en pointillé) approche tout juste les 72 secondes. Le temps référence «POSIX - 1», n'est pas très loin, avec 87 secondes. Enfin, il est important de noter qu'une amélioration du débit de quelques Mo/s a un impact considérable sur les performances. Ainsi, l'approche collective proposée par la bibliothèque ROMIO permet de diminuer le temps pour la granularité de 8 Ko de plus de 80% et de près de 60% pour 32 Ko.

Nous avons décrit les diverses expérimentations menées afin d'analyser le comportement d'une application parallèle fortement dépendante du système de stockage. Plusieurs problèmes évoqués d'un point de vue plus théorique, dans la première partie de ce rapport, ont été vérifiés et validés d'un point de vue expérimental. Dans la suite de ce chapitre, nous présentons les expériences conduites afin d'étudier l'impact qu'ont plusieurs applications entre elles lorsqu'elles sont exécutées de manière concurrente sur une grappe.

4.4. Évaluation multi-applicative

Dans cette section, nous allons étudier l'impact qu'a une application sur une autre. La démarche a été la suivante :

- étude de l'impact d'une application parallèle fortement dépendante du système de stockage sur une application accédant de manière séquentielle à une faible quantité de données ;
- étude de l'impact mutuel de deux applications parallèles fortement dépendantes des E/S ;
- étude des performances de 10 applications (5 dépendantes et 5 moins dépendantes) ayant des comportements parallèles et séquentiels.

Ces tests nous ont permis de montrer l'importance de gérer les E/S d'une manière globale à l'architecture d'exécution. Le temps de complétion de chacune des applications a été mesuré pour des granularités de 8, 32, 128 et 512 Ko.

4.4.1. Impact d'une décomposition de 4 Go sur une opération «`cat`-*like*» brève

Dans ce paragraphe, nous allons évoquer l'impact considérable d'une application fortement dépendant des E/S sur un programme faiblement couplé à l'unité de stockage.

Comme précédemment, la décomposition parallèle est réalisée par le jeu de tests `IOR` : 32 processus répartis sur 32 nœuds récupèrent un fichier de 4 Go. Les interfaces `POSIX` (cas 1) puis `MPI I/O` (cas 2) ont été mises en œuvre. L'application «`cat`-*like*» consiste en une lecture séquentielle de 16 Mo à partir d'un nœud utilisant, dans les deux cas, l'interface `POSIX` (le standard `MPI I/O` n'étant pas prévu à cet effet). Les deux applications accèdent au même serveur `NFS`.

Afin de tenir compte du temps nécessaire au déploiement et à la mise en charge du serveur `NFS`, l'application séquentielle est démarrée 15 secondes après le programme `IOR`. Cette expérience a pour but d'étudier les performances qu'aurait une application, faiblement dépendant des E/S, «schédulée» par le gestionnaire de ressource (qui rappelons le, ne tient pas compte du taux de charge du serveur de données).

Chacun des graphiques apparaissant sur la figure 4.9 présente les résultats obtenus pour une granularité d'accès spécifique pour le «`cat`». Les courbes en trait plein correspondent aux valeurs atteignables lorsque l'application `IOR` exploite l'interface `POSIX` ; à l'opposé les courbes en pointillées nous renseignent sur les performances fournies par la stratégie collective de `MPI I/O`.

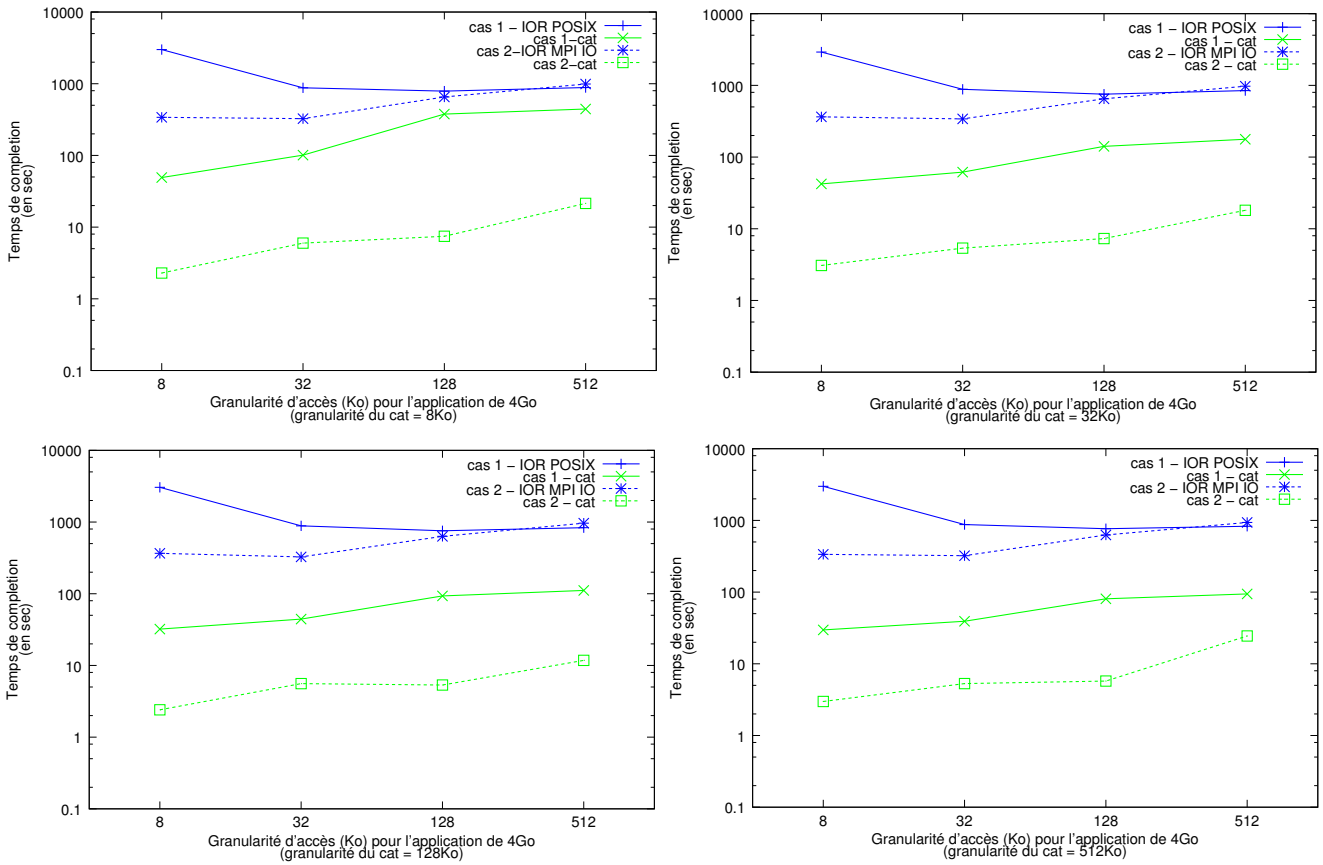


FIG. 4.9 – Impact d'une décomposition de 4 Go sur une opération «`cat`-*like*» de 16 Mo

La première observation concerne le temps nécessaire à l'opération séquentielle pour récupérer le fichier de 16 Mo. Indépendamment de la granularité, le temps requis pour effectuer cette tâche en mono-applicatif est quasi instantanée (moins d'une seconde pour un disque culminant autour de 57 Mo/s et une interface réseau Gigabit). Dans le cas décrit ici, plus la granularité de l'application `IOR` devient grande, plus les dégradations sur les performances de l'application «`cat`-*like*» sont visibles, et ce, sur les quatre graphiques. Lorsque les routines d'accès `POSIX` sont utilisées par le programme `IOR`, le temps nécessaire pour effectuer l'opération de 16 Mo varie d'envion 30 secondes (graphique en bas à droite pour une granularité `IOR` de 8 Ko) à plus de 443 secondes (graphique en haut à gauche pour une granularité `IOR` de 512 Ko).

D'un point de vue plus général, l'application séquentielle profite des petites granularités néfastes pour l'application `IOR` (un nombre moins important de requêtes émanant de la décomposition pendant la durée nécessaire à la récupération du fichier de 16 Mo).

Les temps d'inexploitations engendrées par les mécanismes de synchronisation exploités dans la bibliothèque `ROMIO`, permettent au serveur de satisfaire les demandes de l'application séquentielle. Ainsi, l'impact est moins significatif et les valeurs pour les granularités `IOR` inférieures à 32 Ko sont meilleures (courbes en pointillée). Toutefois, elles restent encore loin des valeurs référence, aussi bien pour l'application de décomposition que pour l'opération «`cat`-*like*». Ces valeurs sont représentées sur les deux graphiques de la figure 4.10. Les courbes exposées au sein de la figure 4.9 ayant énormément de similitudes, nous avons choisi de reprendre seulement les granularités 8 Ko et 512 Ko et avons retiré les mesures correspondant à l'emploi des routines `MPI I/O` afin de ne pas surcharger la figure.

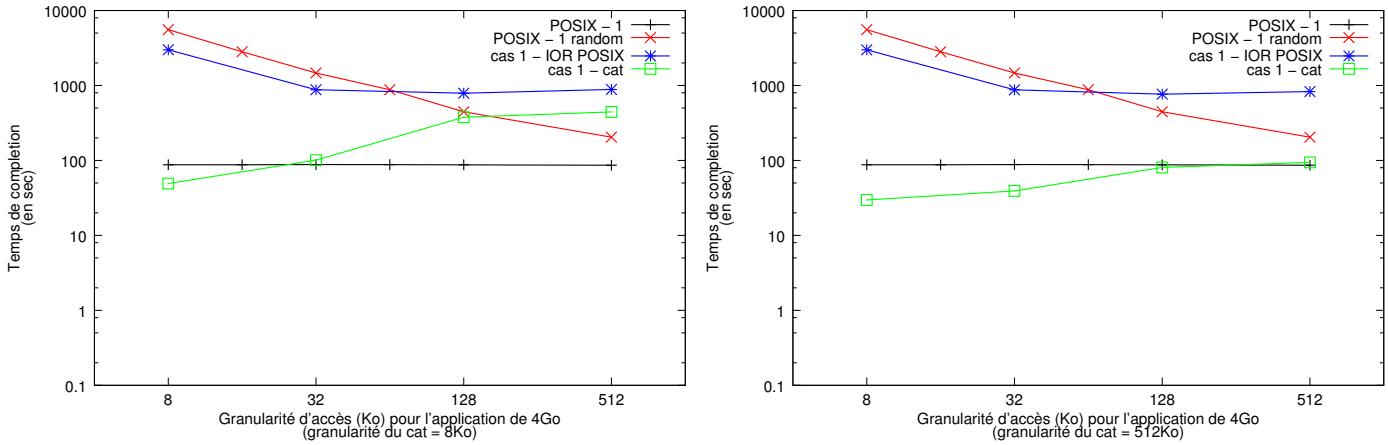


FIG. 4.10 – Impact d'une décomposition de 4 Go sur une opération «`cat`-*like*» de 16 Mo
Comparaison avec l'approche séquentielle et sérialisée. Les deux granularités extrêmes du `cat` ont été retenues pour la comparaison.

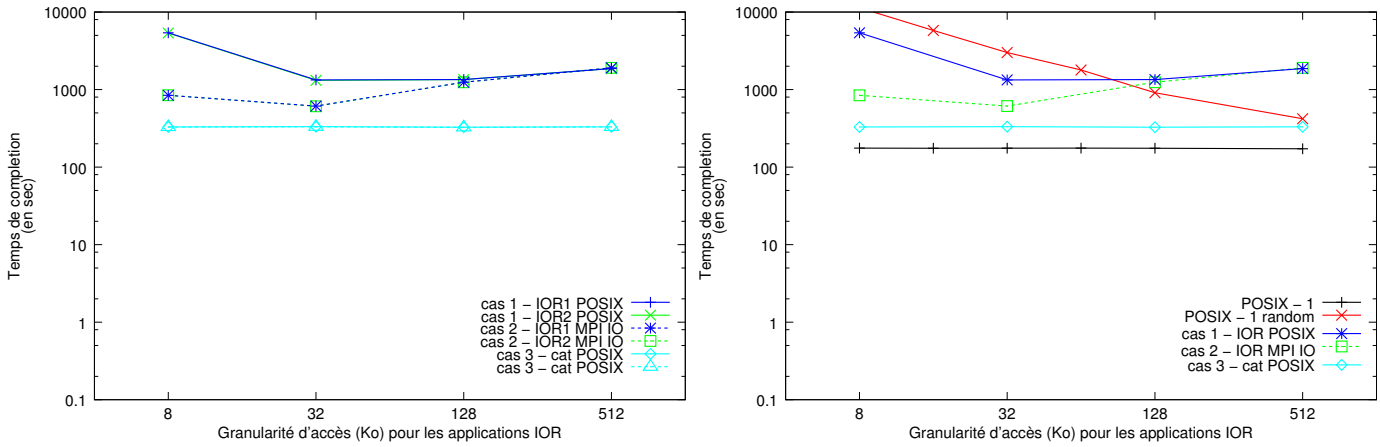
Comme précédemment, les courbes «`POSIX - 1`» et «`POSIX - 1 random`» correspondent aux performances obtenues lors d'une récupération d'un fichier de 4112 Mo (4 Go+16 Mo), en séquentiel et en sérialisé (dans un milieu mono-applicatif, sans concurrence).
Dans les deux cas, la récupération du fichier de 16 Mo nécessite pour les grosses granularités autant de temps que la récupération séquentielle d'un fichier 257 fois plus gros. Pour une granularité d'accès de 8 Ko (graphe de gauche), le temps requis est même supérieur à l'opération sérialisée. L'impact considérable de la décomposition parallèle sur l'opération `cat`-*like* est une nouvelle fois illustré ici.

L'exploitation du concept de sérialisation apparaît une fois encore comme une solution capable de fournir de meilleures performances. Ces deux derniers graphiques ne permettent pas une analyse fine du comportement sérialisé. Le fichier référence de 4112 Mo permet de comparer le temps requis pour récupérer cette quantité en séquentiel et en sérialisé. Toutefois, il est d'ores-et-déjà possible d'entrevoir un modèle sérialisant les accès en provenance de plusieurs applications et à destination d'une unité de stockage. Ainsi, en associant, à cette approche, des contraintes de répartition «équitable» entre les applications, nous devrions continuer à favoriser l'exploitation des caches et limiter le nombre de déplacements.
L'expérience suivante, d'une manière similaire, permet d'étudier l'impact mutuel de deux décompositions.

4.4.2. Impact mutuel de deux applications bornées par les E/S

Nous avons montré l'impact significatif d'une décomposition de 4 Go sur une opération séquentielle de 16 Mo. Le temps nécessaire à la récupération des 16 Mo est égal (voire supérieur) à la lecture d'un fichier de 4 Go. Dans cette section, nous nous intéressons à l'influence qu'ont deux décompositions de 4 Go entre elles. Chacune des deux opérations est réalisée par 32 processus répartis sur 32 nœuds. La granularité d'accès des deux applications `IOR` croit en parallèle. Enfin, un serveur `NFS` dédié, exporte les deux fichiers de 4 Go.

Le graphique à gauche sur la figure 4.11, présente les temps de complétion de chacune des applications : pour le cas 1, les routines `POSIX` pour l'application `IOR 1` et l'application `IOR 2` ; pour le cas 2, les routines collectives `MPI I/O`. Un troisième cas , correspondant à la lecture séquentielle de deux fichiers de 4 Go par deux programmes `cat`-*like* indépendant a également été évalué. Il révèle l'impact de deux lectures séquentielles concurrentes sur des fichiers de taille importante. Le recouvrement des courbes, pour les trois modes d'accès, indique qu'aucune des deux applications n'est défavorisée : le système semble donc être «équitable».



Comparaison `POSIX` vs `MPI I/O` — Comparaison avec les courbes «références»

FIG. 4.11 – Impact mutuel entre deux décompositions de 4 Go
2*(32 procs sur 32 nœuds)

Dans les deux premiers cas, une dégradation significative des performances apparaît très distinctement. L'exploitation en concurrence d'approche basée sur un chargement complet du fichier (cas 3) semble plus performante quelque soit la granularité : autour de 300 secondes alors que les décompositions en parallèle requièrent pour `POSIX` au mieux 1330 secondes et 600 secondes pour `MPI I/O`.

Sur la partie de droite de la figure, nous comparons ces valeurs aux courbes qui font office de référence : la lecture séquentielle, «POSIX - 1», puis sérialisée «POSIX - 1 random» d'un fichier de 8 Go ($4Go * 2$). Les courbes pour les trois cas ont été regroupées pour faciliter la lecture des valeurs. L'approche sérialisée permet à nouveau d'atteindre des meilleurs temps pour des grosses granularités que les comportements parallélisés.

Le temps nécessaire à l'accomplissement de la récupération concurrente des deux fichiers est d'environ deux fois celui requis pour la lecture séquentielle mono-applicative. Le déplacement entre les deux fichiers peut expliquer cette perte de performance. Dans le cas illustré, cette approche est plus performante que la sérialisation. En effet, la lecture des deux fichiers étant séquentielle (et contiguë), seul des déplacements d'un fichier à un autre affectent les performances, aucun déplacement «interne» ne survient (modulo la granularité NFS). Si le nombre de cat en parallèle devenait plus conséquent, le nombre de déplacement possible augmenterait et les performances diminueraient.

D'une manière générale, la réalisation de deux décompositions en parallèle dégrade considérablement les performances. Les débits proposés par l'interface POSIX culminent aux alentours de 2 Mo/sec et ceux de MPI I/O à 5 Mo/sec au maximum pour une unité de stockage capable d'extraire de manière séquentielle plus de 57 Mo/s. Avant de conclure cette étude préliminaire, nous avons souhaité observer les performances susceptibles d'être atteintes au sein d'une grappe lorsque 10 applications indépendantes sont exécutées. Nous allons voir que la concurrence affecte l'ensemble des applications indépendamment de la quantité de données demandées et que les performances globales sont insatisfaisantes.

4.4.3. Haut niveau de concurrence : 10 applications

Un des principaux but de cette thèse est d'étudier la concurrence existant au niveau des systèmes de stockage au sein d'un environnement multi-applicatif. Après avoir analysé certains cas de manière distincte, nous nous attarderons dans cette section sur une dernière expérience couvrant un large éventail des comportements pouvant survenir sur une grappe pour une période.

Le nombre d'applications, les modes d'accès (lecture/écriture), la quantité de données manipulées ainsi que les granularités d'accès ont été arbitrairement retenus. Le but a été de produire un jeu d'évaluation correspondant à une charge potentielle d'une grappe : 10 utilisateurs exécutant différents programmes. Les quatre premières applications accèdent aux données en parallèle et les 6 suivantes de manière séquentielle.
Le jeu de tests a requis 94 nœuds clients et un serveur NFS exportant 10 fichiers pour une quantité totale de 6 Go. Comme précédemment, les applications séquentielles sont lancées avec une retard de 15 secondes afin de permettre le déploiement des programmes parallèles. Chacun des fichiers en écriture a été pré-alloué. Le tableau 4.2 présente un bilan des temps nécessaires pour réaliser chacune des opérations.

Dans un premier temps, considérons ce jeu de tests comme une unique application manipulant des données en lecture et en écriture sur 6 Go. Le temps requis pour terminer cette expérience (correspondant au temps de complétion le plus long dans chacune des colonnes) est d'environ 595 secondes pour POSIX et 840 pour MPI I/O.
L'utilisation, au sein des applications parallèles, de routines fournies par la bibliothèque ROMIO semble augmenter considérablement le temps de complétion pour chacune d'entre elle. Au sein des expériences basées sur l'interface POSIX, il est intéressant de remarquer que cette borne su-

Numéro de l'application	Description	Temps de complétion NFS	
		POSIX	MPI I/O
1	read decompos.. - 2 Go (32 nœuds, granularité=128 Ko)	490	840
2	write decompos. - 2 Go (32 nœuds, granularité=128 Ko)	409	815
3	read decompos. - 256 Mo (16 nœuds, granularité=8 Ko)	595	728
4	write decompos. - 128 Mo (8 nœuds, granularité=64 Ko)	51	257
5	read sequentiel - 1 Go (1 nœud, granularité=2 Mo)	558	59
6	write sequentiel - 512 Mo (1 nœud, granularité=2 Mo)	192	71
7	read sequentiel - 32 Mo (1 nœud, granularité=4 Ko)	531	9
8	write sequentiel - 32 Mo (1 nœud, granularité=4 Ko)	208	9
9	read sequentiel - 4 Mo (1 nœud, granularité=32 Ko)	57	1.5
10	write sequentiel - 4 Mo (1 nœud, granularité=32 Ko)	39	2

Les temps sont exprimés en secondes.

TAB. 4.2 – Performances des E/S sur un jeu de tests de 10 applications indépendantes
Etude de l'impact et comparaison entre l'interface POSIX et les routines collectives MPI I/O.
(94 processus déployés sur 94 nœuds, un serveur NFS dédié)

périeure ne correspond pas à l'application qui a manipulé le plus d'informations. En effet, ici la lecture de 256 Mo par 16 processus (application 3) requiert 10% de temps supplémentaire qu'une lecture séquentielle de 1 Go (application 5) et 20% de plus qu'une lecture ou une écriture d'un fichier de 2 Go (application 1 et 2). Le système qui nous apparaissait équitable lors de la section 4.4.2, se révèle être en fait, avec un nombre plus important d'applications, totalement inefficace selon ce critère. Les valeurs mesurées avec l'interface MPI I/O, même si elles sont importantes, sont cependant en adéquation avec la quantité de données manipulées par les applications parallèles (applications 1 à 4).

Enfin, le dernier point que nous souhaitons évoquer, concerne les valeurs pour les accès purement séquentiels (applications 5 à 10). Lorsque l'ensemble des applications exploite les routines read ou bien write, l'impact est tel que 57 secondes sont nécessaires pour récupérer 4 Mo ou encore 39 secondes pour les écrire (applications 9 et 10). Avec l'utilisation des fonctions MPI I/O, les barrières de synchronisation au sein des applications parallèles permettent aux applications séquentielles de profiter des créneaux d'inexploitation et donc d'être traitées (cf. section 4.4.1).

D'un point de vue général et pour les deux modes d'accès, les valeurs sont largement plus élevées par rapport aux capacités de l'unité de stockage. L'écart entre les mesures est également déplorable : l'approche collective *MPI I/O* requiert plus de 800 secondes pour récupérer 2 Go, alors que pendant la même période moins de 60 secondes sont nécessaires pour rapatrier 1 Go sur un nœud. La mise en œuvre d'une solution ayant pour but une gestion globale des E/S est donc primoriale dans un système multi-applicatif destiné au calcul intensif.

4.5. Bilan

Nous avons vu l'impact d'une décomposition parallèle sur les performances d'un service de stockage et ce même si les programmes exploitent les routines `MPI I/O`. L'étude a été conduite autour du système de fichiers `NFS` car ce dernier est encore largement employé pour le partage des données sur les architectures de type grappe. Les diverses expérimentations nous ont amené à étudier les performances fournies par une approche sérialisée.

Nous avons vu qu'il était possible, en s'appuyant sur ce mécanisme, de maintenir un niveau de performance plus élevé pour des granularités d'accès importantes. La mise en place d'un point de centralisation permettant la régulation de l'ensemble des interactions des E/S de la grappe semble intéressante à analyser. De plus, une sérialisation globale devrait permettre de réordonner certaines requêtes afin de maximiser la contiguïté.

Le schéma 4.12 illustre ces propos : afin de maximiser l'utilisation de l'unité de stockage, il est nécessaire qu'au moins une requête soit présente lorsque la demande précédente se termine. L'approche généralement utilisée est l'envoi au plus tôt de l'ensemble des demandes en attente, entraînant comme nous l'avons montré, des dégradations significatives au niveau des performances. L'idée ici consiste à transmettre, «juste à temps», une requête et seulement une.

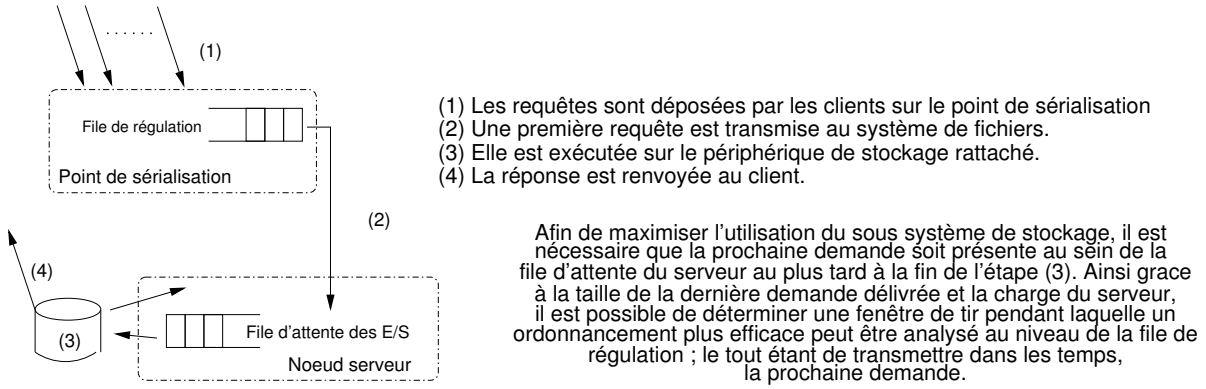


FIG. 4.12 – Sérialisation et ordonnancement

Ainsi, pendant le temps où les accès sont temporisés, il est possible d'appliquer divers politiques de réordonnancement permettant de se rapprocher du comportement séquentiel et contigu le plus performant pour une unité.

Un service basé sur ces concepts devrait être capable de fournir des performances comprises entre les deux valeurs qui nous ont servi de référence tout au long de ce chapitre : l'approche séquentielle «`POSIX - 1`» et celle sérialisée (sans ordre) «`POSIX - 1 random`» (cf. figure 4.13).

De même, en cas de concurrence entre plusieurs applications, il sera possible de définir des stratégies d'équilibrage en adéquation avec les besoins des applications ou des critères plus spécifiques (priorité, ...).

Dans le chapitre suivant, nous abordons les contraintes d'un modèle reposant sur la sérialisation afin de gérer les E/S de manière globale dans une environnement multi-applicatif

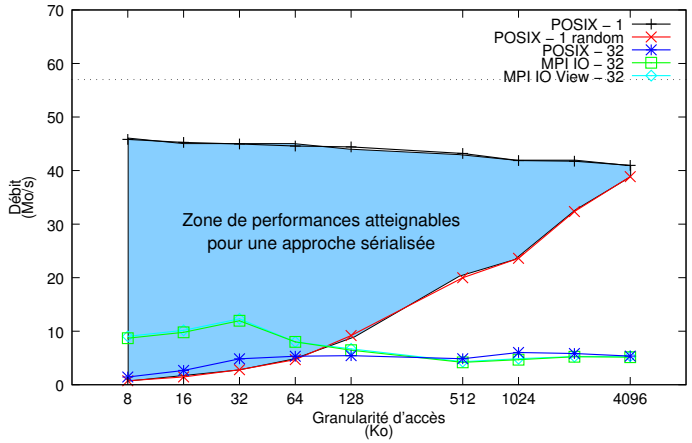


FIG. 4.13 – Performance atteignable par une approche basée sur la sérialisation

comme l'est une grappe. Nous allons étudier dans un premier temps les différents interactions pouvant survenir et présenterons par la suite une stratégie d'ordonnancement global conçue pour optimiser les performances tout en équilibrant l'accès aux ressources de stockage entre les différentes applications.

PRÉSENTATION DU MODÈLE 5

Dans les chapitres précédents, nous avons abordé la problématique des E/S au sein des architectures informatiques. Les dégradations de performances significatives lors de modes d'accès parallèles ou concurrents ont été abordées. Les solutions disponibles comportent plusieurs faiblesses qui limitent les gains de performances dans un contexte fortement concurrent. De plus, elles requièrent des interfaces spécifiques qui en complexifient leur utilisation.

L'étude préliminaire, présentée lors du dernier chapitre, a révélé les gains susceptibles d'être apportés par une solution basée sur un modèle sérialisé et la possibilité d'appliquer des stratégies de réordonnancement. Nous allons décrire et justifier dans ce chapitre la politique de contrôle, de réordonnancement et de régulation des E/S que nous souhaitons proposer.

Dans un premier temps, une étude des interactions des E/S susceptibles d'intervenir au sein d'une grappe va nous permettre, d'une part, d'aborder les contraintes d'une mise en place d'un point de sérialisation et, d'autre part, de synthétiser un grand nombre d'informations recueillies lors de réflexions sur un éventuel modèle d'ordonnancement.

Après avoir rappelé plusieurs notions fondamentales de la théorie de l'ordonnancement, nous montrerons pourquoi l'ordonnancement global des E/S au sein d'un environnement multi-applicatif distribué est un problème complexe à modéliser et à résoudre. L'«*ébauche*» d'un premier modèle sera donné. Le but n'est pas d'exprimer de manière formelle un modèle d'ordonnancement précis mais plutôt de nous positionner par rapport à des problèmes connus et donc nous guider dans les choix d'une stratégie globale.

La seconde partie de ce chapitre présentera l'algorithme retenu dans notre solution. Certaines caractéristiques plus spécifiques seront détaillées et viendront compléter cette descrip-

tion. Les implémentations et les évaluations menées autour de cette méthode seront décrites dans les deux chapitres suivants (cf. chapitres 6 et 7).

5.1. E/S et architecture *HPC* distribuée

Cette section analyse, étape par étape, les différents composants et les contraintes présentes au sein d'une architecture distribuée de type grappe.

Pour rappel, une grappe est un ensemble de nœuds (mono ou multi-processeur) permettant l'accès, à moindre coût, à une forte puissance de calcul. Même si chacun des nœuds possède un espace de stockage et peut par conséquent participer activement au stockage global de l'architecture, une grappe est généralement modélisée par deux groupes : les nœuds dit de «calcul» et les nœuds dit de «stockage» (cf. figure 5.1).

Dans un contexte d'applications parallélisées, fortement dépendantes des E/S, nous avons identifié [1] plusieurs points de contention pouvant survenir :

- côté client, losrque plusieurs processus accèdent à la pile des E/S,
- côté serveur, lorsqu'une quantité importante de requêtes doit être traitée,
- sur les unités de stockage qui repositionnent sans cesse les têtes de lecture.

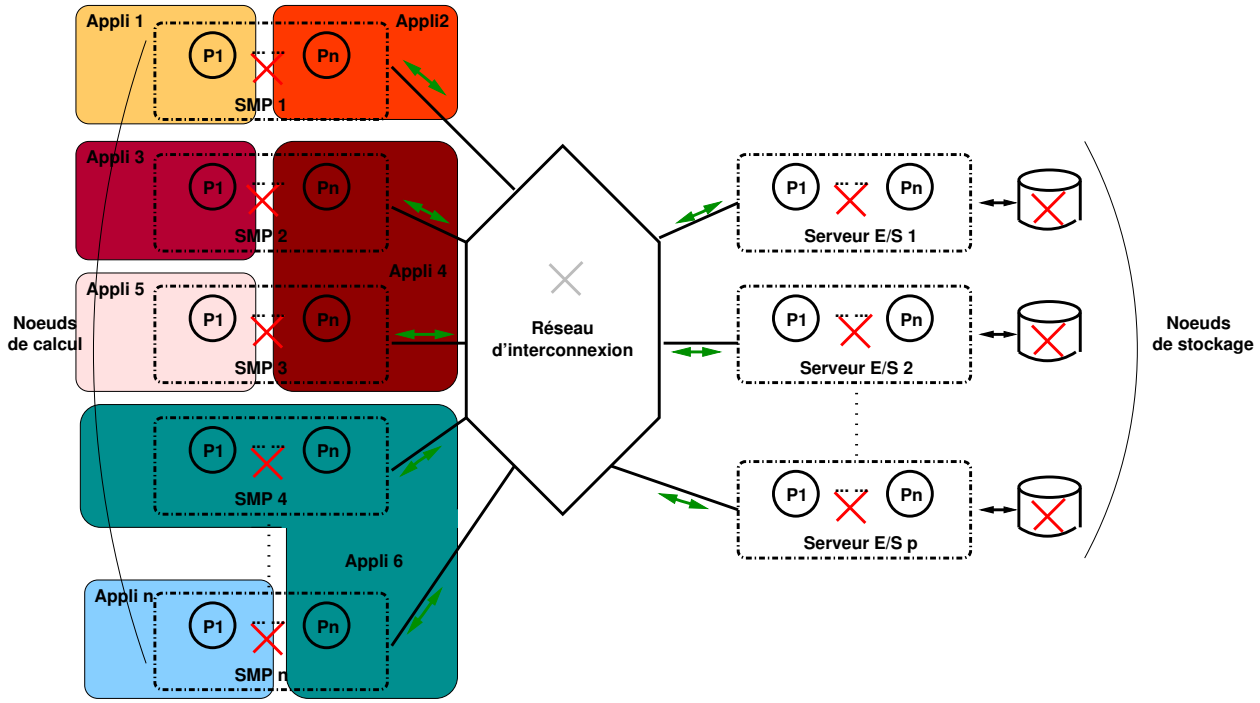


FIG. 5.1 – Grappe et environnement multi-applicatif
Pour faciliter la lecture de la figure, les nœuds de «calcul» sont placés à gauche et ceux de «stockage» à droite. Les croix modélisent les différents points de contention.

[1] D'un point de vue théorique dans le chapitre 2 puis expérimental au sein du chapitre 4.

Comme nous l'avions indiqué, des phénomènes de congestion peuvent également apparaître sur le réseau. Néanmoins, les dernières évolutions technologiques, comme les technologies Myrinet ou Quadrics [Gog05], ont permis de fournir des débits considérables (l'interconnexion GigaEthernet, de plus en plus employée, permet de fournir théoriquement 125 Mo/s) avec des latences très faibles. De plus, la taille des fonds de paniers [2] des matériels actifs récents est définie de manière à supporter de manière simultanée des communications point à point soutenue entre l'ensemble des entités reliées. De par la différence de débit et de latence entre les supports réseaux et les supports de stockage, et afin de simplifier notre modèle, nous supposerons par la suite que le réseau n'est pas un élément limitant de notre système. Pour illustrer ce choix, l'interconnexion GigaEthernet de la grappe de Sophia-Antipolis offre un débit point à point de plus de 100 Mo/s pour une latence entre deux nœuds inférieure à 0.1ms. En comparaison, les unités de stockage présentes sur ces mêmes machines fournissent un débit de 57Mo/s pour un temps d'accès moyen de 8ms. Enfin, il nous paraît nécessaire de souligner les limites de validité de cette hypothèse. En effet dans un contexte grille, les latences réseaux deviennent significatives (de l'ordre de 17ms entre deux sites de l'architecture Grid 5000). Dans un tel contexte, il sera donc primordial d'intégrer cette métrique au sein de notre modèle.

L'approche, que nous souhaitons mettre en œuvre, consiste à exploiter un point de sérialisation permettant de réordonnancer et réguler les E/S de l'ensemble des applications. Notre principal objectif est l'amélioration des performances globales du système de stockage tout en maintenant un certain niveau d'équité et d'interactivité entre l'ensemble des applications. Une stratégie performante mais naïve consiste à regrouper l'ensemble des requêtes par application afin de traiter, par la suite, chacun de ces sous-groupes séquentiellement. Dans une telle approche, les mécanismes de pré-chargement (*read-ahead*) sont largement favorables mais ceci se fait au dépens de la réactivité (interactivité) et peut même créer de la famine [3] . Pour avoir un temps de réponse acceptable et proposer une stratégie équitable, l'accès au système de stockage doit être donné de manière régulière à chacune des applications, cassant ainsi la contigüité dans les accès nécessaires pour atteindre les débits maximum. Nous rappelons qu'un grand nombre de changements ne permet pas une utilisation efficace des caches et engendre éventuellement des repositionnements des têtes de lecture/écriture trés coûteux.

Dans les paragraphes suivants, nous allons nous concentrer sur la mise en place d'un tel point de sérialisation afin d'optimiser les performances au sein d'une application parallélisée puis à un niveau global pour l'ensemble des applications. Les remarques précédentes vont nous permettre de nous positionner par rapport aux différents problèmes d'ordonnancement étudiées dans un cadre théorique.

5.1.1. Sérialisation mono-applicative

Une des étapes importantes dans la proposition d'une stratégie consiste à définir le critère à optimiser. Dans un premier temps, l'efficacité (ou le débit global) de l'application, va nous guider pour établir une politique adéquate. Les techniques de réordonnancement et d'agrégation au sein d'un programme ont été largement étudiées. L'idée principale consiste à essayer de rétablir un ordre séquentiel plus performant (cf. section 2.2.1). Toutefois, la parallélisation des entités de stockage a complexifié la mise en œuvre de ces mécanismes. Pour illustrer ces propos,

[2] Zone temporaire équivalent à un cache assurant l'interconnexion de tous les éléments du commutateur.

[3] Une requête déposée dans la queue peut attendre indéfiniment si des accès avec une priorité plus importante sont transmis au système.

nous allons nous concentrer sur une application parallélisée déployée sur une machine puis sur plusieurs nœuds. La première considération à prendre en compte concerne la fréquence et la régularité dans l'arrivée des requêtes.

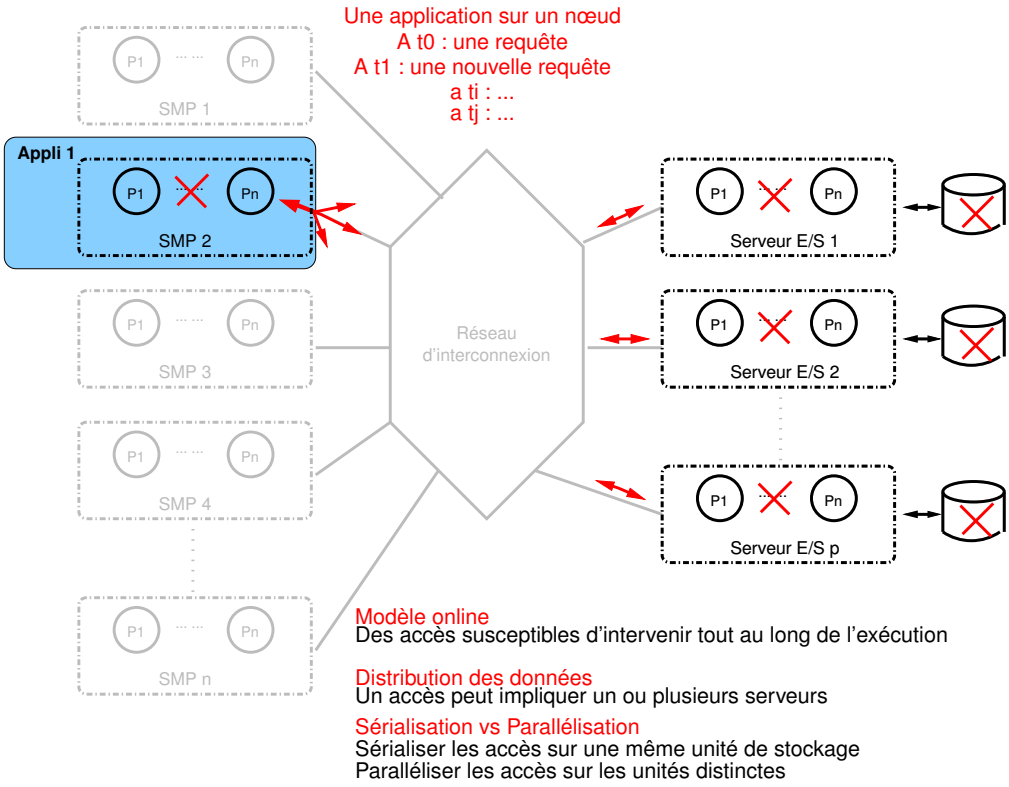


FIG. 5.2 – Modèle *online* et distribution des accès

Chacun de ces accès peut entrainer une requête sur un ou plusieurs serveurs d'E/S selon la granularité de répartition des données (cf. figure 5.2). Chacun des serveurs peut traiter ces requêtes indépendamment. Il existe donc deux niveaux de granularité : la granularité d'accès vue depuis le client et celle employée par le protocole du système de fichiers sous-jacent.

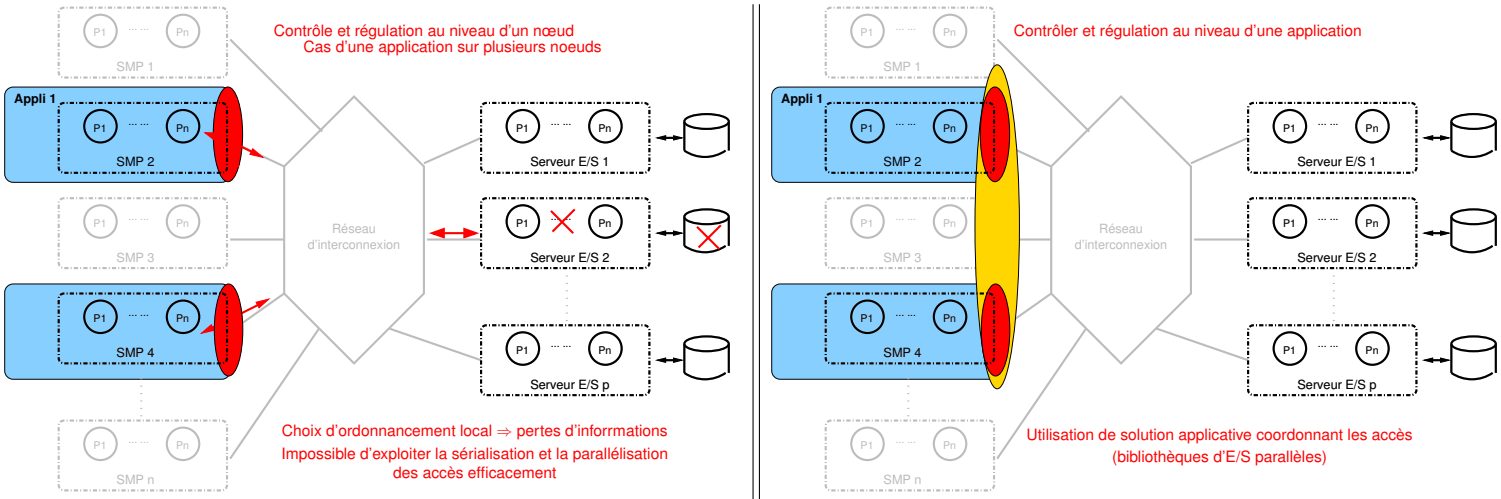
En placant un point de sérialisation au niveau d'un nœud, il est possible d'optimiser les accès aux différents serveurs [4] . La première optimisation consiste en une ré-organisation séquentielle associée à un processus d'agrégation des requêtes (niveau client) afin de maximiser les performances. Après cette éventuelle phase, une prise en compte de la distribution des requêtes clients par rapport à la granularité du protocole du système de fichiers permet une régulation efficace : deux accès sur deux entités distinctes doivent être traités de manière parallèle alors que deux accès sur un même serveur doivent être sérialisés afin de minimiser les déplacements (cf. section 4.5).

Toutefois, cette approche «mono-nœud» ne permet pas une sérialisation globale dans le cas d'une application accèdant aux données depuis plusieurs nœuds. La figure 5.3 (a) illustre ces propos : deux accès à destination d'un même nœud engendrent une dégradation sur l'unité

[4]Cette approche a d'ailleurs été implantée dans un prototype en espace utilisateur et nous a permis de valider notre approche [LD05b].

de stockage s'ils ne sont pas transmis dans le bon ordre. Dans un tel cas, il est par exemple nécessaire d'exploiter un service réparti (figure 5.3 (b)) permettant de contrôler, réordonnancer et réguler les E/S sur les différentes entités de stockage. Ce genre d'approche est généralement employé par les solutions applicatives de type bibliothèque.

Néanmoins, comme nous l'avons mentionné à plusieurs reprises, ces approches se concentrent uniquement sur les interactions propres à une application et ne répondent pas à nos attentes. Par ailleurs, nous verrons lors du prochain chapitre (cf. section 6.1) que la mise en place d'une solution applicative requiert des mécanismes de synchronisation afin de pouvoir sérialiser l'ensemble des accès. Ces mécanismes s'avèrent très coûteux lorsque le nombre de messages échangé est important.



(a) - Application multi-nœuds

(b) - Sérialisation mono-applicative

FIG. 5.3 – Interactions des E/S au sein d'une grappe : cas 1 - mono-applicatif

5.1.2. Sérialisation multi-applicative

Une approche côté serveur, permettant de sérialiser l'ensemble des E/S à destination des unités de stockage, semble dans un premier temps envisageable (cf. figure 5.4). Chacune des unités de stockage (serveur d'E/S + support de sauvegarde) doit déterminer un ordre efficace/interactif et équitable parmi les requêtes des différentes applications présentes dans leur file d'attente.

Cependant, cette solution, à *priori* plus simple à mettre en œuvre, ne permet pas de déterminer la meilleure stratégie du point de vue des applications. La figure 5.4 illustre ces propos : 2 applications, A1 et A2 souhaitent récupérer respectivement 15Mo et 10Mo. La répartition des données sur les serveurs est réalisée par bloc de 10Mo. Ainsi, A1 transmet une demande d'accès de 10Mo sur le serveur d'E/S 1 et de 5Mo sur le serveur d'E/S 2. Pour l'application A2, les 10Mo souhaités sont répartis entre deux serveurs, deux requêtes de 5Mo sont émises vers le serveur 1 et 2.

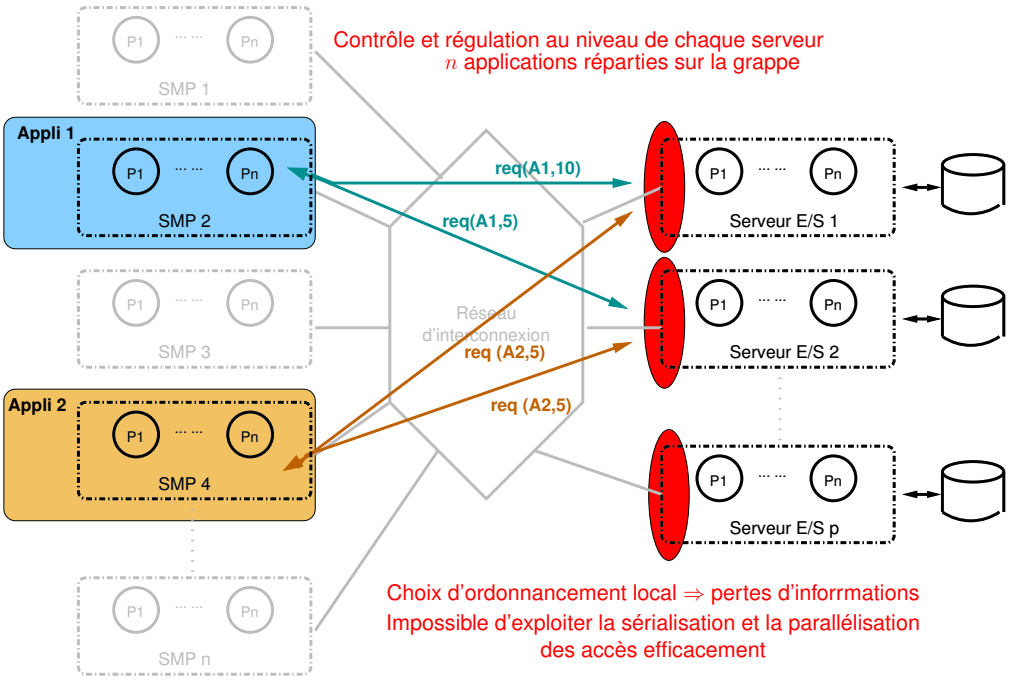


FIG. 5.4 – Interactions des E/S au sein d'une grappe : cas 2 - multi-applicatif

La figure 5.5 présente l'état de chacune des files d'attentes. Le temps nécessaire pour exécuter une requête est fonction de la taille de données manipulées. Ainsi, les temps pour exécuter ces requêtes sont respectivement 10 et 5 unités de temps. Une stratégie «premier arrivé premier servi» réalisée indépendamment sur chacun des systèmes peut pénaliser une des applications. Dans le cas illustré, la requête A1 de 10 unités de temps, sur le serveur d'E/S 1, est traitée avant la requête A2. L'application A2, qui doit attendre la terminaison de ses deux requêtes pour accèder à ses données, est pénalisée et doit attendre 15 unités. Dans une stratégie plus globale, une permutation des requêtes sur le serveur d'E/S 1 améliore le temps de service de l'application A2 de 10 unités de temps en augmentant seulement de 5 unités A1.

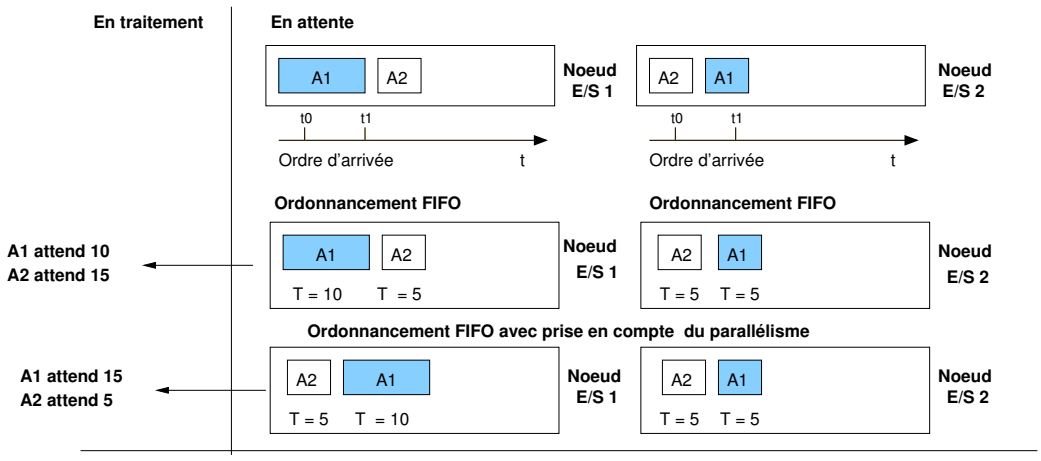


FIG. 5.5 – Interactions des E/S au sein d'une grappe : dépendance entre les applications
De manière similaire à une sérialisation côté nœuds de calcul,
une sérialisation côté serveur ne permet pas de définir la stratégie la plus efficace

L'idée ici n'est pas de mentionner le fait que la stratégie «premier arrivé premier servi» n'est pas efficace dans l'exemple mais plutôt d'introduire la forte dépendance entre les accès applicatifs et les sous-requêtes qui en découlent. Une stratégie ayant pour but de maximiser les performances de chaque application tout en maintenant un niveau acceptable pour l'interactivé et l'équité doit impérativement tenir compte de ces aspects induits par cette double granularité (applications *vs* protocole du système de fichiers) et le facteur de répartition des données. Une synchronisation entre les serveurs semble donc requise et la mise œuvre d'un tel service, sans influer sur les performances apportées par le parallélisme, peut se révéler délicate.

5.1.3. Bilan

Nous venons d'évoquer plusieurs aspects relatifs aux E/S au sein d'une grappe. Plusieurs contraintes ont été mentionnées :

- Les requêtes arrivent de manière indépendante et tout au long de l'exécution de chacune des applications.
- Il est nécessaire le cas échéant d'agréger les accès portant sur un même fichier. Toutefois, il est important de limiter la taille maximale d'agrégation afin de ne pas avoir un impact sur l'interactivité.
- Chaque requête «cliente» peut engendrer plusieurs sous-accès «système» vers les différents serveurs (double granularité).
- La «régulation» doit prendre compte la structure du système de fichiers afin de sérialiser les accès à destination d'une même unité et exploiter la parallélisation dans le cas contraire.
- La double granularité (point de vue client *vs* protocole du système de fichiers) associée au contexte multi-applicatif complexifient la mise en œuvre d'une stratégie globale d'ordonnancement des requêtes. Il est impératif de prendre en compte les relations entre les différents serveurs.

D'un point de vue général, la mise en place d'un point de centralisation semble obligatoire. Néanmoins, comme nous l'avons rapidement abordé, placer un tel service au sein de l'architecture distribuée s'avère compliqué. Afin d'aborder étape par étape notre problème, nous allons nous concentrer dans un premier temps sur une proposition autour d'un unique serveur. Nous reviendrons par la suite sur les contraintes imposées par un système de stockage parallélisé.

Dans la section suivante, nous rappelons plusieurs notions fondamentales de la théorie de l'ordonnancement. Les diverses remarques établies vont, en plus de nous positionner par rapport aux problèmes connues de la thématique, nous guider dans le choix d'une stratégie.

5.2. Vers un modèle d'ordonnancement

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 spatial et temporel des tâches (processus). Cette section tente de formaliser de manière un peu plus rigoureuse notre problème d'ordonnancement. Un premier rappel de plusieurs notions fondamentales d'ordonnancement théorique va nous permettre de classifier notre problème. Le contexte associé à la diversité de nos objectifs, ne nous a permis de définir un modèle précis. L'ébauche que nous avons réalisée est présentée à la section 5.2.3.

5.2.1. Notions fondamentales

Les paragraphes de cette sous section ont pour but de familiariser le lecteur avec certains concepts fondamentaux de la thématique de l'ordonnancement. Elle n'est qu'une très succincte introduction par rapport à la quantité des travaux qui ont été réalisés dans le domaine [LKA04]. Le premier paragraphe introduit la terminologie utilisée pour classifier les problèmes d'ordonnancement. La notation de Graham communément employée y est présentée. Les paragraphes suivants abordent de manière plus détaillée la notion de critère d'évaluation. Un exemple va nous permettre d'illustrer la difficulté dans le choix de cette métrique qui détermine la stratégie à mettre en œuvre. Nous clôturerons cette partie en introduisant une spécificité importante de certaines stratégies d'ordonnancement. Cette caractéristique a été proposée afin de remédier au problème des comportements synchrones dans l'arrivée des tâches. Chacun des concepts introduits au sein de cette section est développé de manière plus approfondie dans [LKA04].

Terminologie

Dans le vocabulaire lié à l'ordonnancement et par analogie, nous dirons qu'une «requête» correspond à une «tâche» (notée j) et que le «serveur de données» correspond à un «processeur» (notée i). Le temps nécessaire au traitement de la requête j sur le serveur i est noté p_{ij} (*Processing Time*). Le temps d'arrivée d'une requête est noté r_j. L'importance associée à une tâche est noté w_j. D'autres notations sont employées (temps de traitement espéré, échéance limite, ...) mais il ne nous a pas semblé intéressant de les présenter ici.

La classification selon Graham est couramment utilisée pour décrire les problèmes d'ordonnancement, elle est définie de la façon suivante :

$$\alpha|\beta|\gamma \tag{5.1}$$

Contraintes de ressource :

Le champ α nous renseigne sur l'architecture d'exécution et ne contient qu'une seule entrée. Il peut correspondre à des valeurs comme : une machine (1), des machines parallèles identiques (P), des machines hétérogènes (Q, chacune des machines a une vitesse s_i différente et $p_{ij} = \frac{p_j}{s_i}$) ou encore des machines parallèles pour lesquelles le temps de traitement d'une tâche sur une machine dépend de ces affinités avec la machine (R, $p_{ij} = \frac{p_j}{s_ij}$). Il existe d'autres possibilités mais là encore nous ne présentons que les valeurs qui nous semblent pertinentes dans notre contexte.

Contraintes d'ordonnancement :

Le champ β fournit des détails sur les caractéristiques des tâches et les contraintes d'ordonnancement. Il peut être composé de plusieurs valeurs. La valeur (r_j) indique, par exemple, que les tâches ne sont pas toutes disponibles au temps 0 et que chacune est caractérisée par une estampille déterminant la date à laquelle elle est entrée dans le système. Les restrictions sur la durée de traitement d'une tâche ($p_j = p$) ou encore l'échéance (d_j) sont d'éventuelles caractéristiques qui peuvent être mentionnées. Les valeurs concernant les contraintes d'ordonnancements indiquent si les tâches peuvent être préemptées ($pmtn$) ou bien si des contraintes de précédences entre les tâches existent ($prec$).

Ce champ permet également d'indiquer si le problème est *offline* ou *online*. Dans un modèle *offline*, tous les travaux sont connus d'avance et le choix d'un ordonnancement efficace pour

un critère donné est facilité. Au contraire, dans un modèle *online*, les tâches (requêtes) arrivent au fur et à mesure et le système d'ordonnancement ne prend connaissance d'une tâche que lorsqu'elle arrive. De plus, les problèmes *online* se divisent en deux sous catégories : *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 connues 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*.

Contraintes de métrique :

Le champ γ indique le critère à optimiser. Les différentes valeurs sont généralement fonction du temps de complétion des tâches (noté C_j le temps de complétion de la tâche j).
Les métriques basées uniquement sur le temps de complétion sont :

- Le *makespan* correspondant à la date de finition de tous les travaux ($\max C_j$).
- La somme (ou totalité) des temps de finition, *total completion time* (ΣC_j).

Dans un contexte comme le nôtre où des dates d'arrivée (r_j) interviennent, les temps de réponse ou encore le *strech* sont plus appropriés que le temps de complétion :

- Le temps de réponse d'un travail est la différence entre son temps de finition (C_j) et son temps d'arrivée r_j (noté $F_j = C_j - r_j$).
- Le *strech* est une mesure pour exprimer le temps d'attente d'un travail (ou d'un utilisateur). Il est définit de manière suivante : $S_j = \frac{F_j}{p_j}$. Cette métrique assez récente [BCM98] permet de pondérer le temps de réponse par rapport à la quantité de travail demandée. Cette pondération permet d'avoir un ordonnancement qui tient compte de la disparité des caractéristiques de chaque tâche. Ce qui peut s'avérer très pertinent dans un contexte où le but est d'optimiser les performances de chacune des applications en tenant compte d'une qualité de service particulière. D'une manière plus générale, elle permet de quantifier le ralentissement que subit une tâche dans le système.

Comme pour le temps de complétion, les stratégies basées sur ces métriques tentent de minimiser soit le maximum soit la somme des F_j ou S_j :

- Le maximum des temps de réponse ($\max F_j$) s'apparente à l'équité. Cela permet d'assurer qu'aucune des tâches ne souffrira de famine.
- La somme des temps de réponse (ΣF_j), cette métrique représente les performances globales du système (temps de réponse moyen pour chacune des applications). La totalité des temps de réponse, *total response time* est connu parfois sous le nom de *total flow time*.
- De manière similaire, le maximum des *stretchs* ($\max S_j$) ou la somme des *stretchs* (ΣS_j) peuvent également être utilisés.

Choix du critère d'évaluation

Le choix du critère est prépondérant lors de la conception d'une stratégie d'ordonnancement. Pour illustrer ces propos, nous allons analyser les différents ordonnancements appliqués par 3 stratégies usuelles [Bak74] [PST04] [KP00].

- *First In First Out*, cette politique, appelée parfois *First Come First Served*, exécute toujours le travail qui est arrivé le premier. De par sa simplicité, elle est souvent mise en œuvre. Elle optimise le $\max F_j$.

– *Shortest Remaining Processing Time»*, à chaque événement nécessitant un processus d'ordonnancement (nouvelle entrée dans le système ou fin de traitement de la tâche en cours), cet algorithme exécute toujours les travaux avec le temps restant le plus petit. Cette métrique optimise la somme des temps de réponse (ΣF_j). Cependant, elle peut créer des processus de famine. Elle garantit également un facteur 2 pour Σ_j.
– *Round Robin*, cet algorithme distribue de manière équitable la ressource à tous les travaux de façon instantanée. En pratique, chaque tâche est exécutée pendant un quantum de temps fixe. La liste des tâches est parcouru de manière circulaire. Chaque nouvelle tâche est placée soit en début soit en fin de liste. Cette stratégie permet un partage instantané des ressources équitable (entre deux événements d'ordonnancement, la ressource a été partagée entre l'ensemble des requêtes présentes dans la queue). Même si la métrique n'a pas été clairement définie, nous associerons cette valeur à la notion d'interactivité.

Dans notre exemple (figure 5.6), 3 requêtes de 3 applications distinctes (A0, A1, A2) sont envoyées à un serveur de données. Les tailles de données de ces requêtes sont 50, 5, 20 Mo respectivement (l'écart entre les valeurs est volontaire, il va permettre de montrer l'importance dans le choix du critère). Supposons que le temps nécessaire pour exécuter une requête (p_j) correspond à la taille de données manipulées. Ainsi, les temps pour exécuter ces requêtes sont respectivement 50, 5 et 20 unités de temps. Enfin, chacune des requêtes est entrée dans le système (estampille r_j) avec un décalage : A0 marque le départ ; par rapport à A0, A1 et A2 arrivent respectivement 10 et 15 unités de temps après.

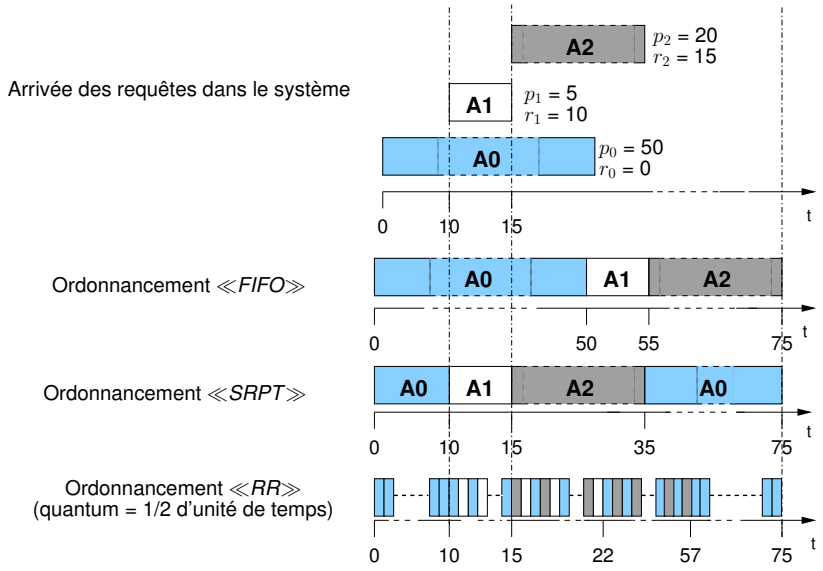


FIG. 5.6 – Importance du critère et de la stratégie d'ordonnancement
Comparaison de 3 stratégies usuelles optimisant chacune un critère différent.

Le tableau 5.1 présente les résultats selon la métrique $\max(F_j)$ et ΣF_j. Nous avons également choisi d'indiquer le *stretch* de chaque tâche pour l'ensemble des stratégies. Comme attendu, les résultats obtenus coïncident avec les remarques réalisées auparavant. Ainsi, l'algorithme *FIFO* minimise les problèmes de famine (critère d'équité min-max, $\max(F_j)$) tandis que la stratégie *SRPT* offre les meilleures performances d'un point de vue global (le temps de réponse moyen est minimisé). D'un point de vue interactivité, la troisième stratégie est la meilleure. Le découpage par quantum permet en effet de servir l'ensemble des tâches dans une période de qn (avec q taille du quantum et n nombre de tâches à servir).

Critère	*FIFO*	*SRPT*	*RR*
Application	$A1/A2/A3$	$A1/A2/A3$	$A1/A2/A3$
F_j	$50/45/60$	$75/5/20$	$75/12/42$
S_j	$1/9/3$	$1.5/1/1$	$1.5/2.4/2.1$
$\max(F_j)$	60	75	75
ΣF_j	165	100	129

TAB. 5.1 – Importance du critère et de la stratégie d'ordonnancement
Valeurs atteintes (en unité de temps) pour chacun des critères en fonction de la stratégie appliquée.
Le ralentissement que subit chaque tâche est exprimé sur la dernière ligne.

Dans une gestion globale des E/S d'une grappe, nous avons identifié nos objectifs comme étant la performance globale du système, l'équité entre les applications (éviter les processus de famine) et un critère d'interactivité défini afin de ne pas pénaliser les applications faiblement dépendantes des E/S. Malheureusement, l'exemple traité nous montre qu'il est impossible d'optimiser ces trois critères à la fois. La stratégie *RR* semble offrir le meilleur compromis performance/interactivité. Néanmoins, dans notre exemple, le coût de changement de contexte est nul or nous avons, à maintes reprises, évoqué l'impact lié à un tel changement (mauvaise utilisation du cache, repositionnement des têtes). Même si cette stratégie semble de manière intuitive répondre en partie à nos objectifs, il est primordial de trouver des méthodes permettant de limiter les basculements d'une application à une autre. Ainsi, efficacité, équité et interactivité pourront être traitées. Nous reviendrons par la suite sur ces aspects.

5.2.2. Ordonnancement oisif

Les stratégies d'ordonnancement employées au sein des unités de stockage exploitent souvent des approches non oisives (*work-conservative*), c'est-à-dire qu'à partir du moment ou au moins une tâche est prête à être exécutée, alors l'ordonnanceur en élit forcément une parmi elles. 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 non oisif ne répond pas forcément à la politique d'ordonnancement souhaitée en plus de dégrader les performances.

La figure 5.7 illustre ces propos. Deux lectures synchrones (`cat`) sont réalisées en parallèle. La stratégie désirée est un ordonnancement de partage proportionnel. Ce type d'ordonnancement partage le disque entre plusieurs applications selon une proportion définie. Par exemple, dans le cas illustré, nous souhaitons une proportion de 3 pour 2 (3 requêtes de l'application 1 pour 2 de l'application 2. Les requêtes arrivant de manière synchrone, un ordonnancement non-oisif va engendrer une stratégie de type *FIFO* et la proportion devient 1 :1. Dans le cas 2, l'approche oisive permet d'attendre l'arrivée de la seconde et de la troisième requête de l'application 1 et fournit l'interactivité attendue.

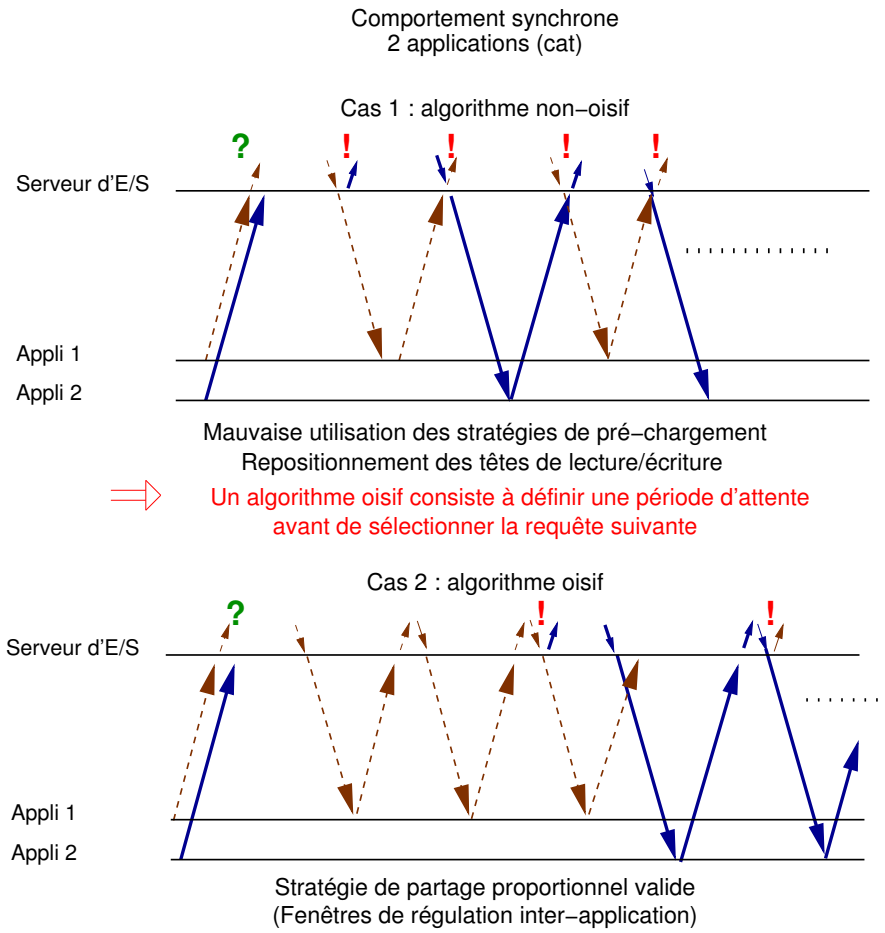


FIG. 5.7 – stratégie non-oisive et oisive
Chaque point d'exclamation engendre une mauvaise utilisation des caches en plus d'un déplacement potentiel. Un algorithme oisif permet de valider la stratégie d'ordonnancement désirée en cas de comportement synchrone.

5.2.3. Ébauche d'un modèle

Nous avons identifié notre problème comme étant *online* : les tâches (les requêtes) continuent d'arriver durant le processus d'ordonnancement et le nombre total de requêtes n'est pas connu d'avance [Ban03].

Par ailleurs, des requêtes sur un même fichier doivent être regroupées si elles sont contigües afin de maximiser les effets des caches et limiter les repositionnements intempestifs. Cette nouvelle tâche, créée lors du processus d'agrégation, a une durée de traitement plus grande (correspondant à la somme des temps de chacune des tâches la composant). En supposant que cette nouvelle requête ne soit pas sélectionnée par la stratégie d'ordonnancement et que d'autres accès pouvant être regroupés soient déposés dans le système, alors la taille (la durée) de la requête est à nouveau modifiée. Ce comportement évolutif dans le temps ne nous permet pas de caractériser si notre problème est clairvoyant ou non. En effet, il est impossible de prévoir à tout moment la durée effective d'une requête. Nous disposons d'une borne inférieure sur la durée minimale (au moment d'une itération) mais aucune information ne peut être donnée concernant la taille future de la requête.

Cependant afin d'éliminer les problèmes de famine, il semble obligatoire de limiter le processus d'agrégation à une taille maximale définie. Nous n'avons pas trouvé de modèle intégrant de tels comportements au sein des problèmes d'ordonnancement connus.

Dans le cas où le système de stockage est parallélisé, les échanges pouvant survenir répondent à un modèle n/n : chacun des nœuds (des clients) peut transmettre n requêtes vers les différents serveurs. Chaque serveur est indépendant et traite les demandes présentes dans sa file d'attente. Le traitement d'une tâche sur un serveur ne peut être préempté. Selon la notation de Graham, abordée lors de la section 5.2.1, l'ordonnancement sur les serveurs s'apparente à *priori* à un problème de type *Job Shop* (*J*). En effet, chacun des accès, d'un point de vue client, génére une ou plusieurs sous-requêtes. Chacune d'entre-elles est alors transmise vers un serveur selon la répartition des données (*data-striping*). Ainsi, une tâche (un accès client) implique un traitement sur un ou plusieurs serveurs.

Au problème *Job Shop*, il faut associer des contraintes concernant le temps de traitement p_{ij}. En effet, une requête va interagir soit avec une zone cache soit avec une unité de stockage. Comme nous l'avons indiqué au début du manuscrit (cf. section 1.1), ces temps sont d'un ordre de grandeur différent. Ainsi le temps de traitement p_{ij} varie entre :

$$tempscachehit \leq p_{ij} \leq tempscachemissavectempscachemiss \gg tempscachehit \tag{5.2}$$

Lors d'accès de type lecture, la probabilité d'avoir les données j dans un des caches dépend des derniers ordonnancements réalisés sur le serveur i : soit l'accès a récemment été servi et les données sont encore dans le cache, soit le mode d'accès correspond à un comportement séquentiel et il est impératif de tenir compte des mécanismes de pré-chargement (cf. section 2.2.3). Ainsi, pour les accès de type lecture, il semblerait qu'une notion d'affinité (ou synergie) apparaisse entre les sous-accès d'un même serveur [5] . Pour les accès de type écriture, l'impact du cache dépend de la politique de gestion choisie (mode synchrone *vs write-behind*). A notre connaissance aucun modèle ne s'apparente à notre problème.

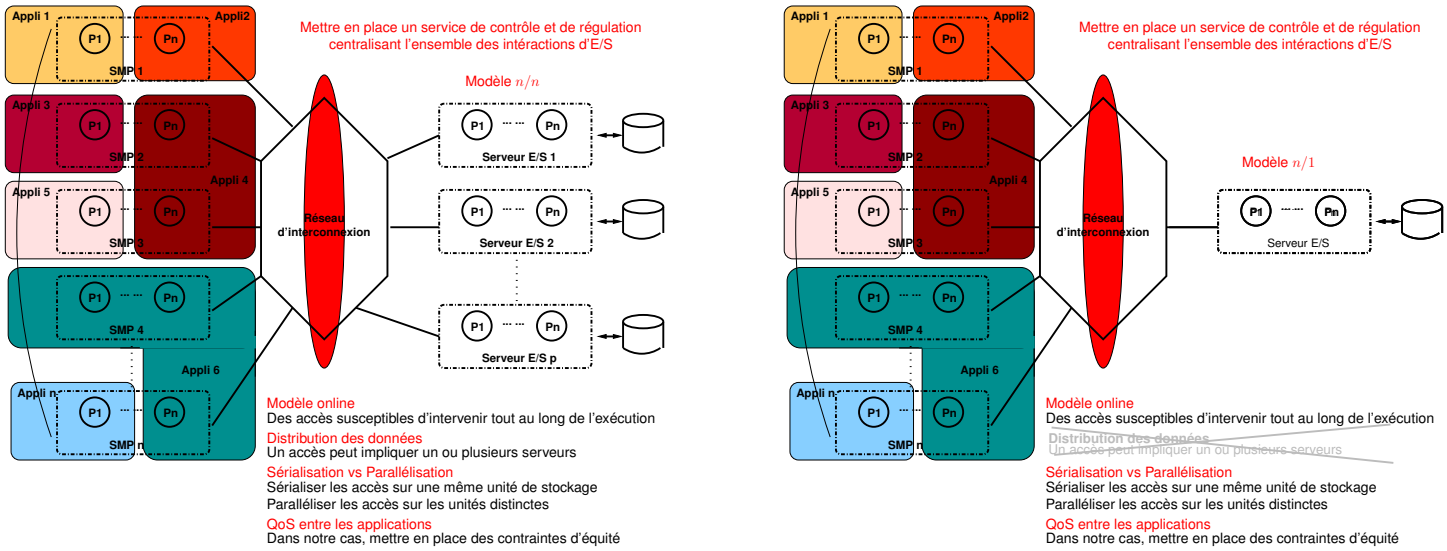
La deuxième difficulté de notre problème réside dans le choix du critère d'évaluation : nous souhaitons d'une part maximiser les performances en agrégeant/combinant les requêtes en de plus larges et d'autre part assurer un minimum d'interactivité entre les applications ainsi qu'un niveau d'équité permettant d'éviter de manière rigoureuse les processus de famine [6] . Le *stretch*, qui nous a paru dans un premier temps une métrique pertinente dans notre contexte, se révèle être difficilement utilisable à cause du comportement évolutif des p_j. Les métriques $\max(F_j)$ et ΣF_j semblent donc plus adaptées. Cependant, la littérature a montré que l'optimisation simultanée de ces trois critères est impossible à garantir [LSV05] [7] . L'utilisation d'heuristique au cas par cas semble donc être obligatoire. La figure 5.8 présente cette première ébauche.

La description formelle de l'ensemble des interactions E/S émanant d'applications parallélisées à destination de plusieurs serveurs est un modèle qui n'a jamais été étudié à notre connaissance. Le travail décrit ici consiste en une première esquisse qui a pu être réalisée après

[5] Le terme de *précédence* souvent employé ne correspond pas ici. Il n'y a pas d'obligation au sens fort comme il est possible de trouver dans certains problèmes.

[6] De manière intuitive, l'équite correspond à offrir des performances identiques à chacune des applications. Toutefois, cette équité (définie sous les termes d'*équite Max-Min*) correspond à un type d'équité particulier. Il existe en effet plusieurs façons de la définir [MW00]. Dans notre cas, c'est surtout la notion de famine qui nous intéresse.

[7] D'un point de vue plus formel, il a été montré que $\max(F_j)$ et ΣF_j ne peuvent déjà pas être optimisés simultanément.



(a) - Système de fichiers parallèle

$J|online, r_j|\Sigma Fi : fairness, interactivity$

(b) système de fichiers centralisé

$1|online, r_j|\Sigma Fi : fairness, interactivity$

FIG. 5.8 – Ordonnancement global au sein d'une grappe
Afin de parvenir à contrôler, réordonnancer et réguler les E/S, il est nécessaire d'exploiter un point qui va centraliser l'ensemble des accès (modèlisé ici par l'élipse).

un grand nombre d'échanges et de recherches avec plusieurs personnes expertes du domaine. Il nous a permis de déterminer une classe d'algorithme sur laquelle nous avons pu proposer diverses heuristiques.

5.2.4. Bilan

Nous avons vu que la modélisation précise de notre problème n'est pas immédiate et peut devenir rapidement complexe. Afin de résoudre étape par étape nos objectifs, nous avons choisi de nous concentrer dans un premier temps sur une proposition autour d'un unique serveur (figure 5.8 (b)). Ainsi, nous allons proposer une stratégie capable de répondre au critère d'évaluation que nous avons défini comme étant : l'efficacité sous contraintes d'interactivité et d'équité. Cette heuristique est décrite dans le paragraphe suivant. Sa mise en œuvre a été réalisée puis évaluée au-dessus des architectures à base de système de fichiers centralisé à la `NFS`. Comme nous l'avons montré lors du précédent chapitre, un tel serveur doit supporter un nombre conséquent d'accès simultanés, ce qui constitue un cadre tout à fait approprié pour évaluer l'intérêt d'un service d'ordonnancement global.

Nous avons choisi de mettre en place une politique d'ordonnancement essayant de minimiser la totalité des temps de réponse tout 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 majorité des cas à des fichiers spécifiques). L'heuristique que nous avons retenue est décrie dans la section suivante.

5.3. Stratégie d'ordonnancement proposée

Comme nous l'avions vu lors de la section 5.2.1, un algorithme à base de quantum semble pouvoir répondre à nos objectifs. Cependant, nous avions noté l'importance de prendre en compte le coût induit par un «basculement» d'un fichier à un autre. La stratégie *Multilevel Feedback* est une variante des algorithmes à base de quantum [KP97] [CGKK04]. Largement décrite dans [Tan01], cette méthode est généralement celle employée dans les systèmes d'exploitation pour l'ordonnancement des processus. Elle utilise un ensemble de queues ayant chacune un degré de priorité et un temps de traitement spécifié. Chaque fois qu'une nouvelle tâche est créée, elle est insérée dans la queue qui a la priorité la plus élevée. Si le temps alloué lors de son premier passage n'est pas suffisant, elle est alors insérée dans une queue de priorité moindre mais de temps de traitement plus élevé. Les queues sont traitées selon leur ordre de priorité. Une queue ayant la propriété basse est traitée seulement quand toutes les queues qui ont les propriétés les plus élevées sont vides. Ce système de queue à différents niveaux peut permettre de diminuer le nombre de «basculement» coûteux tout en continuant à répondre aux aspects interactivité/performance. Néanmoins, comme une grande partie des systèmes à base de niveaux, cette stratégie peut engendrer des famines. En tenant compte de cet aspect et en nous appuyant sur le concept de quantum à taille variable, nous avons développé une heuristique.

5.3.1. Une variante de l'algorithme *Multiple Level Feedback*

L'algorithme que nous proposons a été conçu pour optimiser le temps moyen de réponse tout en supprimant les problèmes de famine. Pour cela, il attribue à chaque requête en attente un quantum qui croît dans le temps.

Il peut être décrit de la manière suivante :

1. Les requêtes déposées par les clients sont triées par fichiers puis par type (une queue pour les lectures et une pour les écritures) et enfin par leur *offset* de départ et leur taille. Le fait d'ordonnancer les requêtes sur ces critères différencie notre solution des ordonnanceurs de plus bas niveau qui s'appuient principalement sur des blocs disque et des numéros de secteur (cf. section 1.2.2.3).
2. Chaque requête se voit attribuer un quantum initial de zéro.
3. Le processus d'agrégation est réalisé sur les différentes queues. Chaque file d'attente est parcourue selon l'ordre des *offsets* et les accès contigus sont agrégés. Le quantum de la nouvelle requête ainsi réalisée (nommée requête «virtuelle») correspond à la somme des quantums individuels.
4. Le quantum de chaque requête est alors augmenté d'une valeur fixe QB de taille plus ou moins significative selon le degré d'interactivité voulu.
5. La première requête qui possède un quantum assez grand pour lui permettre d'être traitée est sélectionnée (selon l'ordre des *offsets* au sein d'un fichier et selon l'ordre *FIFO* entre deux fichiers). De manière à ne pas avoir d'impact sur la cohérence, chaque requête est estampillée lors de son arrivée. Cette technique permet de garantir un ordre strict : si une écriture est en attente d'exécution pour un fichier donné, toute lecture sur ce même fichier avec une estampille plus récente ne pourra être retenue par l'algorithme d'ordonnancement.

D'une manière générale, comme le quantum croît dans le temps au sein de l'approche *MLF*, les petites requêtes sont temporairement prioritaires. De ce fait, les requêtes virtuelles de taille plus importante (générées par le processus d'agrégation) restent plus longtemps dans le système favorisant ainsi de nouvelles agrégations (en d'autres termes, favorisant l'optimisation du débit). Cette approche permet d'obtenir un temps de réponse satisfaisant pour les tâches interactives. Par ailleurs, une taille maximale pour les requêtes virtuelles est utilisée afin de ne pas agréger de manière infinie et de rencontrer des problèmes de famine. Ainsi, le système permet de maintenir une répartition décemment équitable entre tous les fichiers. Il est possible de favoriser un critère au dépend d'un autre. Par exemple, avec un quantum de taille très grande, le débit peut être considérablement amélioré mais au dépend de l'interactivité et de l'équité. Il est également possible d'envisager une qualité de service paramétrable selon les applications en définissant un quantum spécifique par programme.

La figure 5.9 illustre le comportement de l'algorithme pour deux processus (A1 et A2) accèdant au fichier 1 (fd=1) de manière parallèle sur une base de découpage (*stride*) de 128Ko et deux processus (A3 et A4) accèdant aux fichiers 2 et 3 séquentiellement (une requête après l'autre). Dans l'exemple, tous les accès portent sur 32Ko ; QB est fixé également à 32Ko.

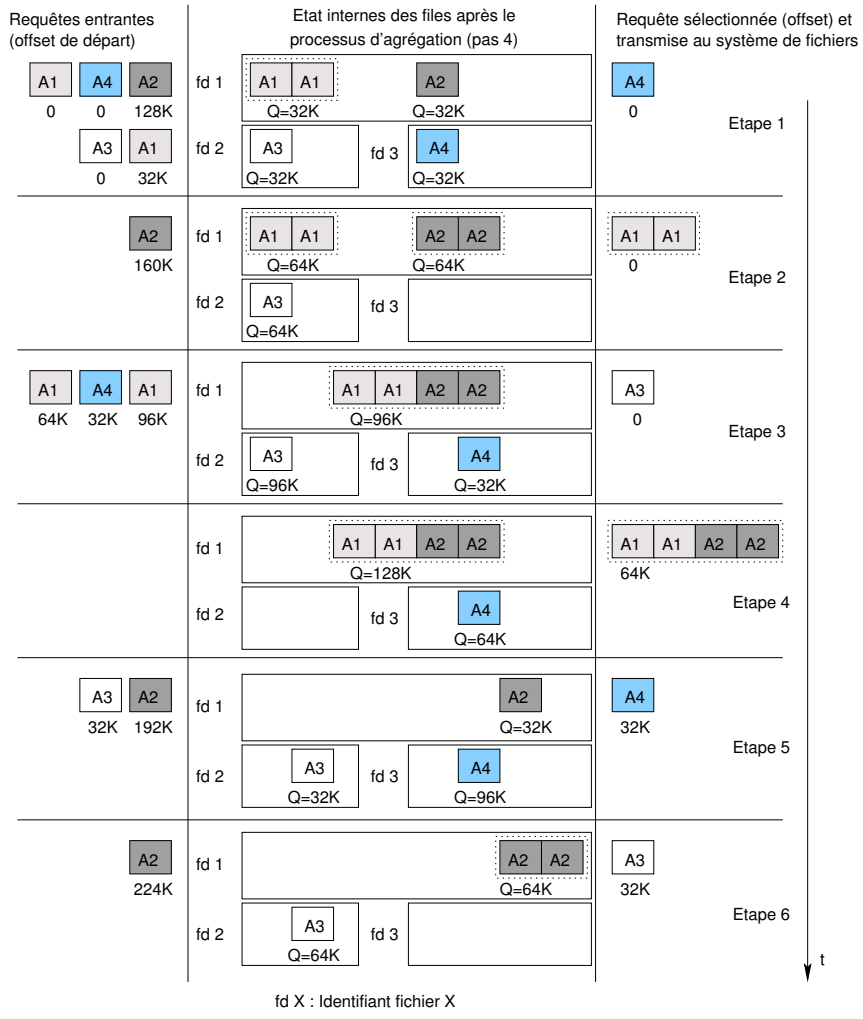


FIG. 5.9 – Une variante de *Multiple Level Feedback*

À l'étape 1, les requêtes de A2, A3, et A4 répondent toutes au critère de sélection (le quantum proposé est assez grand). La requête A4 ayant été déposée avant les deux autres demandes, elle est sélectionnée. À l'étape suivante, la nouvelle requête A2 est agrégée. Une nouvelle fois, chacun des accès peut être sélectionné, l'ordre *FIFO* est appliqué et la requête virtuelle de l'application A1 est ordonnancée. À l'étape 3, seules les requêtes A3 et A4 sont prises en considération, les requêtes agrégées A1/A2 nécessitant un quantum supérieur. Selon l'ordre *FIFO*, la demande A3 est transmise au serveur. L'algorithme continue d'être appliqué sur les étapes 4, 5 et 6.

5.3.2. Optimisation spécifique

Par cette stratégie d'ordonnancement, nous espérons pouvoir détecter les comportements parallèles au sein d'un même fichier afin de reproduire un mode d'accès séquentiel bien plus performant. Néanmoins l'ordre d'arrivée des diverses requêtes étant dépendant de plusieurs facteurs (avancement de chacune des instances dans le programme, latence réseau, ...), il est nécessaire d'apporter quelques améliorations à la variante *MLF* définie préalablement. De plus, il est impératif d'intégrer à la stratégie une approche oisive lui permettant de limiter les effets néfastes induits par des comportements purement synchrones (cf. section 5.2.2). Nous abordons ces deux points par la suite.

Détection des schémas d'accès parallèles

L'ordre et la périodicité d'arrivée des requêtes en provenance des différents processus sont prépondérants dans la variante *MLF* présentée préalablement.

Une des premières difficultés pour parvenir à agréger les requêtes émanant des différents processus d'une application parallélisée, consiste à corriger le décalage induit par le déploiement de l'application. Comme nous l'avons signalé (cf. section 4.3.1), les lanceurs exploités pour déployer des applications parallèles à grande échelle engendrent un décalage plus ou moins important entre le début de chaque instance. Ce décalage est généralement corrigé tout au long de l'exécution du programme par des barrières de synchronisation. Un des objectifs de notre proposition est de ne pas faire appel à des mécanismes analogues qui sont coûteux.

Nous avons légèrement modifié l'algorithme afin d'essayer de favoriser des points de jonction 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 afin de lire un même fichier (`read(10,20)`, `read(20,30)`, `read(30,40)`, `read(40,50)` [8]).
Les requêtes `read(30,40)` puis `read (10,20)` arrivent au sein de la file d'ordonnancement 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), le critère de sélection basé sur le quantum QB est appliqué indépendamment sur les deux accès.
Dans ce cas précis, elles ont toutes deux les mêmes caractéristiques et la requête `read(30,40)` étant arrivée en première, est exécutée. Lors de la prochaine itération, la queue contiendra, deux requêtes contigües et une requête disjointe : `read(10,20)` (précédemment en attente) et les nouvelles requêtes `read(20,30)` et `read(40,50)`. Si au contraire la requête ayant un *off-*

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

set plus petit, `read(10, 20)`, avait été choisie à l'étape 1, la queue contiendrait les requêtes `read(20,30)`, `read(30,40)` et `read(40,50)` qui pourraient être agrégées en une requête `read(20,50)`. Dans le premier cas, trois repositionnements ont été réalisées et les techniques de pré-chargement ont été exploitées une seule fois (entre `read(10,20) et read(20,30)`). Dans le second cas, un seul déplacement est nécessaire au pire (pour se positionner au début du fichier) et les techniques de pré-chargement sont exploitées pleinement.

En s'appuyant sur ces observations, il suffit de choisir la requête ayant l'offset le plus petit parmi les requêtes disponibles tout en tenant compte des problèmes de famine. Nous avons modifié le critère de sélection *FIFO*, employé lorsque deux requêtes d'un même fichier satisfont le critère de sélection, en un critère basé sur l'offset (la plus petite valeur étant prioritaire modulo une échéance limite fixée).

Cette première modification de l'algorithme permet à l'ensemble des processus d'avancer dans la manipulation des données de manière similaire. Toutefois, elle ne permet pas de résoudre un second problème qui a un impact tout aussi important sur les performances. Lorsqu'un processus a un léger décalage et qu'il transmet une requête trop tard pour profiter du processus d'agrégation, il va pénaliser l'ensemble des performances tout au long de l'opération parallèle.

La figure 5.10 illustre ce phénomène : quatre processus participent à la décomposition d'un même fichier.

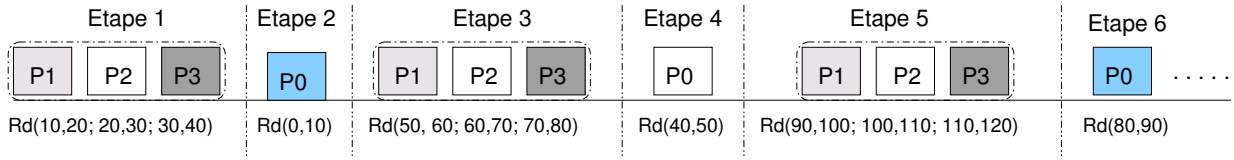


FIG. 5.10 – Décalage récurrent
L'ordre d'arrivée des messages a un impact sur le processus d'agrégation. Dans cet exemple, la requête P_O n'est jamais agrégée.

A l'étape 1, la requête du processus P_0 arrive en retard par rapport aux autres requêtes (P_1,P_2, P_3). Les requêtes de P_1, P_2, P_3 sont contiguës et sont par conséquent agrégées au sein d'une même requête «virtuelle». Quand la requête P_0 arrive au sein de la file d'ordonnancement la requête «virtuelle» a déjà été délivrée. La demande P_0 est insérée dans la file d'attente. Elle est exécutée à l'étape 2. A l'étape 3, les nouvelles requêtes de P_1, P_2 et P_3 arrivent et sont de nouveau agrégées. La demande de P_0 qui arrive une fois de plus en retard ne peut être agrégée. Le scénario se répète alors. Les accès en provenance de P_0 ne pourront jamais être agrégés et le schéma global d'accès ne sera jamais détecté. Ce phénomène est d'autant plus amplifié que la périodicité d'envoi du processus P_0 est longue.

Comportement synchrone

Comme nous l'avons présenté lors de la section 5.2.2, un algorithme non oisif peut rapidement devenir inapproprié en présence de comportements synchrones concurrents. Nous avons complété la stratégie proposée précédemment afin qu'elle puisse traiter, de manière efficace, les comportements synchrones susceptibles d'intervenir. De manière analogue à l'algorithme *anticipatory scheduling* [ID01] (cf. section 1.2.2.3), un bref délai offre au serveur la possibilité de

recevoir la requête suivante et ainsi continuer à traiter de manière séquentielle les accès au sein du même fichier. Cela peut facilement être mis en place au sein de la stratégie décrite en proposant un quantum plus grand que celui uniquement requis pour la complétion de la requête. Le temps supplémentaire ainsi alloué permet d'attendre la nouvelle requête et le cas échéant de bénéficier des stratégies de pré-chargement. Le nombre de «basculement» d'un fichier à un autre (et le coût associé) est ainsi également réduit.

Cette première amélioration permet d'intégrer à notre stratégie l'approche oisive requise afin de remplir notre objectif d'efficacité sous contraintes d'équité et d'interactivité. Cependant, elle n'est pas suffisante pour résoudre de manière fine deux comportements synchrones très distincts : une manipulation synchrone sur une très grande quantité de données (de l'ordre du Mo et du Go) et l'accès sur des portions infimes (de l'ordre de l'octet voir du Ko).

- Manipulation sur une grande quantité de données : dans le cas d'accès conséquent, la stratégie d'un quantum croissant de manière linéaire en fonction de QB n'est pas adaptée. En effet, les requêtes étant transmises de manière synchrone, un processus récurrent proposant un quantum toujours de même taille va apparaître tout au long de la manipulation. Dans un tel cas et usuellement pour ce fichier, il serait plus avantageux de fournir un quantum supérieur à chaque itération tant que celui ci est totalement exploité. Par ce quantum spécifique, la stratégie pourrait exploiter au mieux la contiguité entre l'ensemble des accès.
- Accès sur des portions infimes : dans le cas d'accès portant sur de faibles quantités de données, le quantum alloué peut s'avérer beaucoup trop important (particulièrement pour les accès de faible taille qui ont été déposés dans les queues depuis plusieurs itérations). Le temps excédant se révèle alors inutile et devient un surcôut non négligeable lorsque le nombre d'accès tombant dans cette classe devient important.

Pour traiter de manière efficace, ces deux modes d'accès, nous proposons de prendre en compte l'historique des accès sur le fichier et d'adapter la taille du quantum en conséquence. Pour chaque fichier ouvert (lecture et/ou écriture), une valeur associée nous informe sur le taux d'utilisation du dernier quantum proposé. Si le taux est important, la stratégie délivre un quantum supérieur. Au contraire, si le taux est faible, il est diminué. Enfin, dans le cas où le taux est relativement proche de zéro, les informations sur l'historique du fichier sont réinitialisées (fin de fichier) et le quantum de base QB est réalloué lors de l'accès suivant.

Cette approche à base de quantum dynamique permet d'allouer des «fenêtres d'accès dédiées» (ou «fenêtre de régulation») à chacune des applications s'exécutant sur la grappe. La taille de la fenêtre est définie selon les besoins de l'application modulo les bornes qui assurent l'interactivité et l'équité au sein du système.

5.4. Bilan

Dans ce chapitre, nous avons abordé d'un point de vue théorique la mise en place d'un point de sérialisation au sein d'une architecture distribuée multi-applicatif de type grappe. Ce point va permettre, en plus de réguler les accès, d'appliquer une stratégie de réordonnancement globale pour l'ensemble des E/S émanant des applications en cours d'exécution.

Les différentes recherches conduites au sein de la thématique d'ordonnancement (cf. section 5.2) nous ont montré :

– La difficulté de fournir un modèle précis recouvrant l'ensemble des contraintes propres à notre contexte (double granularité : application *vs* protocole du système de fichiers, comportement évolutif de la taille des requêtes, ...).
– L'impossibilité de garantir les 3 critères souhaités, à savoir l'efficacité, l'équité et l'interactivité. Ce dernier critère n'étant, à notre connaissance, pas encore défini de manière rigoureuse au sein de la thématique. Dans notre cas, nous l'avons défini comme un partage instantané des ressources équitable.

Toutefois, les observations établies nous ont permis de faire plusieurs analogies avec des comportements bien connus. A titre indicatif, l'ordonnancement sur un système de fichiers semble par exemple s'apparenter à un problème de type $JobShop$.

Afin d'aborder étape par étape notre problème, nous avons choisi de concentrer nos efforts sur la proposition d'un service autour d'un unique serveur. Nous reviendrons très succinctement sur les contraintes imposées par un système de fichiers parallèle lors du bilan général (cf. chapitre 8). Le critère d'évaluation retenu a été l'efficacité sous contraintes d'équité et d'interactivité. Les algorithmes à base de quantum semblent les plus adaptés dans notre cas. Ainsi, nous avons proposé une stratégie basée sur l'algorithme *Multiple Level Feedback* (*MLF*). Notre variante intègre un processus d'agrégation des accès contigus au sein du même fichier. La tâche traitée (une requête ou un agrégat de requêtes) est sélectionnée en fonction d'un quantum dynamique qui varie selon le temps d'attente au sein de la queue. Le temps alloué est fonction d'un historique associé à chacun des fichiers manipulés. Cette approche à base de quantum permet de répartir l'accès à la ressource entre toutes les requêtes qui sont présentes dans le système.

Par ailleurs, le fait d'ordonnancer les requêtes sur ces critères plus applicatifs comme le descripteur de fichier, la taille ou encore l'*offset* différencie notre solution des ordonnanceurs de plus bas niveau qui s'appuient principalement sur des blocs disque et des numéros de secteur. Ce changement introduit un niveau d'abstraction supplémentaire dans les techniques d'ordonnancement des E/S offrant ainsi la possibilité de mettre en place des stratégies plus fines.

Dans le chapitre suivant, nous allons décrire les deux prototypes qui ont été réalisés afin d'intégrer la stratégie proposée au sein d'un service disponible sur une grappe. Le premier repose sur un modèle client/serveur indépendant. Le second est un module noyau *Linux* pouvant être interconnecté à une large gamme de système d'E/S. Les avantages et les inconvénients de ces deux approches y sont traités.

IMPLANTATIONS DES PROTOTYPES 6

Les objectifs définis à la fin de la première partie de ce mémoire (cf. section 3.3) consistent en la proposition d'un service capable d'améliorer les performances des E/S d'une manière globale à l'ensemble des applications s'exécutant sur une grappe.

Les principales caractéristiques évoquées étaient :

- l'exploitation d'algorithmes d'E/S parallèles permettant d'améliorer les performances ;
- l'utilisation unique des routines usuelles `POSIX` (`open/read/write/lseek/close`) afin d'assurer la simplicité et la portabilité de la proposition ;
- une coordination globale de l'ensemble des E/S émanant de la grappe.

Les deux derniers chapitres ont permis d'affiner ces objectifs. Tout d'abord, lors de l'étude préliminaire, nous avons vu que l'utilisation indépendante de stratégies d'algorithmes d'E/S parallèles était inappropriée dans un contexte multi-applicatif (section 4.4.3). Ainsi, l'exploitation d'algorithmes doit impérativement être associée à la coordination globale des accès.
En s'appuyant sur le concept d'une sérialisation globale des accès qui s'est révélé plus opportun, nous avons construit une stratégie d'optimisation globale décrite au chapitre précédent.
Les divers aspects abordés lors de cette proposition ont révélé la difficulté dans le choix d'un critère d'évaluation. En effet, l'optimisation unique du critère de performance n'est pas appropriée dans un contexte multi-applicatif. Il est impératif de trouver un bon compromis entre efficacité, équité et interactivité (cf. section 5.1).
Ainsi, en plus des notions d'agrégation et de recouvrement utilisées dans les bibliothèques parallèles, la politique globale d'ordonnancement permet de maintenir une équité et un temps de réponse acceptable pour l'ensemble des applications.

Dans ce chapitre, nous présentons deux implantations qui permettent d'appliquer de manière transparente la variante de l'algorithme *Multiple Level Feedback*. Ces deux prototypes sont très différents et ont en commun uniquement la transparence d'utilisation du point de vue du code. C'est-à-dire qu'aucune routine spécifique à notre solution n'est visible au sein des applications.

Développés tout deux en langage C, le premier repose sur un modèle applicatif client/serveur non-intrusif. Malheureusement, comme nous allons l'aborder, plusieurs contraintes complexifient la mise en œuvre de cette première approche et les évaluations qui ont été réalisées non pas été concluantes [Van05] [LDV05]. Cependant, nous avons souhaité présenter les différentes analyses qui ont été menées autour de ce prototype. Tout d'abord, elles nous ont largement aidé lors de mise en place d'une seconde proposition. Ensuite, elles ont soulevé plusieurs problèmes intéressants susceptibles d'être de bonnes perspectives pour des travaux futurs.
Plus intrusive d'un point de vue système, la seconde solution implantée est un module noyau offrant un service d'ordonnancement générique pour des systèmes d'E/S *GNU/Linux*. C'est la solution *aIOLi* [1] .

Les caractéristiques techniques de chacune de ces deux implantations sont détaillées au sein de ce chapitre.

6.1. *aIOLimaster*, vers une solution transparente et non intrusive

La première implantation réalisée a eu pour but de proposer une solution transparente et non-intrusive basée sur un modèle client/serveur, similaire à l'approche *PANDA* [SCJ+95]. Nous la décrivons dans cette section.

6.1.1. Centralisation des accès

La stratégie que nous souhaitons mettre en œuvre nécessite de connaître l'ensemble des interactions présentes au sein de l'architecture distribuée. Cependant à la différence d'un supercalculateur où la totalité des interactions transite par un unique noyau, une grappe n'offre généralement pas une vision globale de l'ensemble des ressources et des interactions entre les différents nœuds. La mise en place ou l'exploitation d'un point central au sein de l'architecture est alors impérative.

Plusieurs topologies peuvent être alors envisagées. De par la littérature, nous savons qu'un unique point central permet certes de simplifier le modèle mais comporte plusieurs défauts aussi bien au niveau de la tolérance aux pannes (*Single Point of Failure*) qu'au niveau du passage à l'échelle. Cette solution envisageable sur des architectures de petite et moyenne taille nécessite toutefois de bien avoir conscience de ces caractéristiques.
Une topologie hiérarchique corrige en partie ces défauts en répartissant la charge sur plusieurs niveaux. Cependant un point central (le service au plus haut niveau) reste toujours critique en cas de panne.
La figure 6.1 illustre cette notion.

[1] Le terme *aIOLi* vient d'un tout premier prototype, *an Input/Output LIbrary*. La connotation avec le plat méditerranéen nous faisant toujours autant sourire, nous avons choisi de le garder.

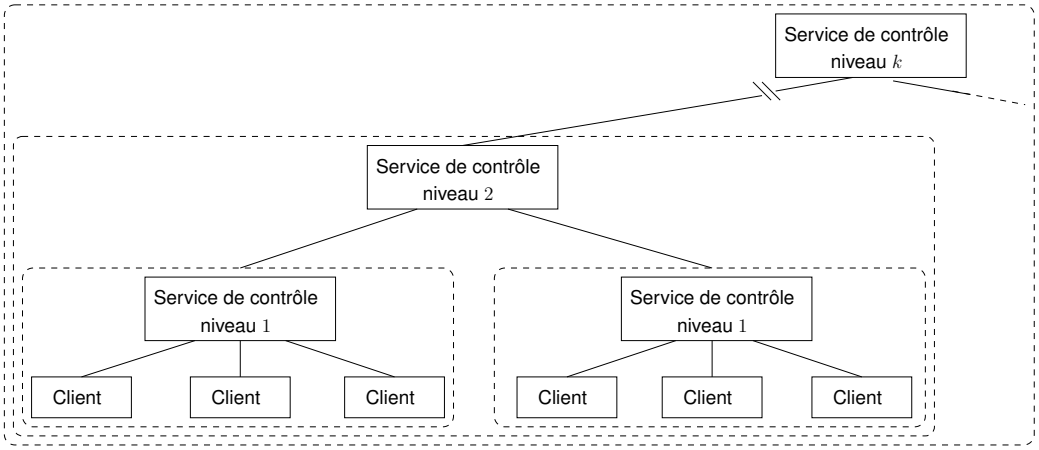


FIG. 6.1 – Centralisation hiérarchique

A chaque niveau de la structure, le service de contrôle transmet la requête qui lui semble la plus pertinente au niveau supérieur. Chaque service de contrôle d'un niveau k est géré par un niveau supérieur $k + 1$. Le service présent au plus haut niveau délivre la requête retenue au système de stockage. Le nombre de niveaux est dépendant de la taille du système. Cette approche semble tout à fait adaptée aux architectures de type grappe de grappes où la passerelle pourraît par exemple servir de nœud de contrôle pour les requêtes à destination d'autres architectures.

Le terme «client» apparaissant au niveau le plus bas de la structure a été choisi volontairement. Il est en effet difficile de déterminer qui joue ce rôle. Nous avons vu lors du chapitre précédent qu'il n'est pas possible d'associer un nœud à un service de contrôle. En effet, cette approche ne permet pas de tenir compte des accès émanant des autres processus de l'application parallélisée lorsque celle-ci est déployée sur plusieurs nœuds (cf. section 5.1.1). La seconde possibilité se rapporte aux solutions applicatives type bibliothèque : chacune des applications coordonne ses propres requêtes et transmet la plus pertinente à un service désigné comme maître.

Dans cette solution, chaque application parallèle appartient à un sous-groupe qui est géré par un service de contrôle. Chacun des services de contrôle transmet alors la requête qu'il lui semble la plus pertinente au service supérieur. Ce dernier applique à son tour la stratégie d'ordonnancement.

Cette architecture logicielle, qui paraît intéressante à première vue, repose sur une hypothèse contestable : chaque application accède à des fichiers qui lui sont propres. Si cela ne se vérifie pas, un service d'optimisation comme celui-ci ne sera pas en mesure de prendre les meilleures décisions en temporisant de manière distincte des requêtes à destination d'un même fichier. De plus, comme nous allons l'aborder dans le paragraphe suivant, il aura potentiellement un impact sur la cohérence puisqu'il lui sera impossible de garantir un ordre entre les accès de type lecture et écriture.

En s'appuyant sur ce constat, il paraît alors judicieux d'associer le terme «client» à celui de fichier. L'ensemble des accès pour un même fichier est transmis à un service d'optimisation. Une hiérarchie s'appuyant sur une notion de groupe de fichiers est construite au-dessus de la même manière que précédemment. La principale difficulté de cette solution est de déterminer

comment et où les accès pour un fichier doivent être transmis. Nous reviendrons par la suite sur cette approche. Elle constitue une des perspectives à moyen terme des travaux autour de la solution *aIOLi*.
Durant la suite, nous définirons un client comme étant une application parallélisée souhaitant accéder au système de stockage.

6.1.2. Agrégation et maintien de la cohérence

De manière analogue à la centralisation implicite des interactions autour d'un noyau, l'agrégation des accès et le maintien de la cohérence se complexifient en milieu distribué. Au sein d'un unique nœud, la cohérence et la gestion des différents niveaux de caches est à la charge du noyau. Ainsi, l'agrégation de plusieurs accès au sein d'un même nœud (figure 6.2) n'a pas d'impact sur la cohérence lors d'opérations en écriture intervenant à *postériori*.

Dans l'exemple, une application parallélisée, constituée de 4 processus, s'exécute sur un même nœud ; 4 requêtes de type lecture à destination du même fichier transitent par le service de régulation présent sur le nœud. Les accès y sont agrégés et une unique demande est transmise au serveur de stockage (étape 1). La réponse est renvoyée au nœud, insérée dans le cache si besoin est, puis, déposée dans l'espace mémoire du service de contrôle. Ce dernier redistribue alors les données à chacun des clients. Si une écriture intervient par la suite, les modifications sont propagées au sein du cache et vers le serveur si nécessaire. Les données restent intègres.

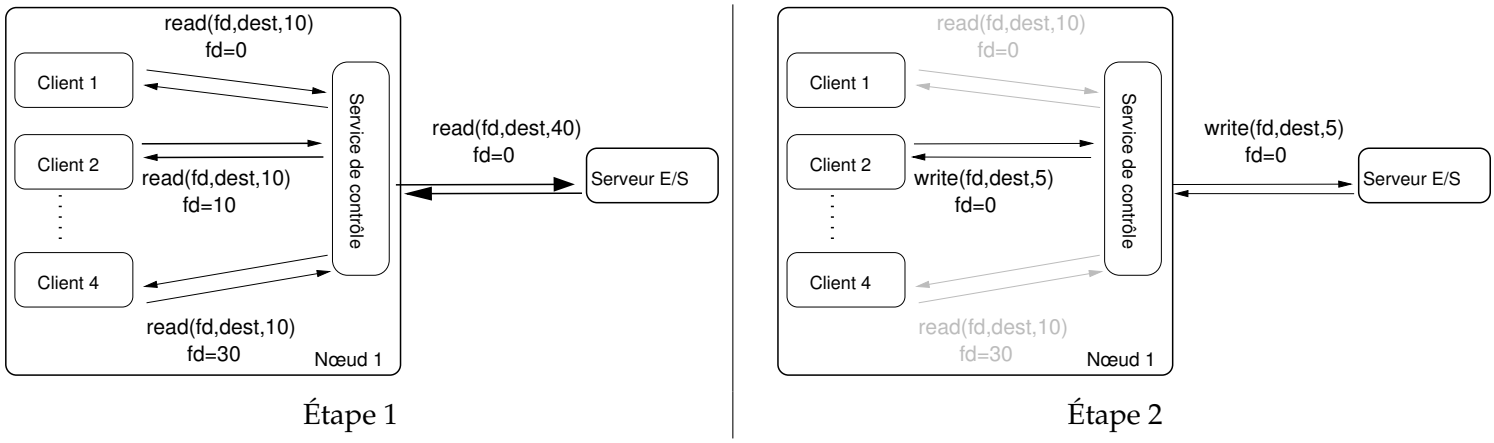


FIG. 6.2 – Agrégation et cohérence au sein d'un nœud
fd correspond au decripteur du fichier. Il contient la position de l'accès dans le fichier (fd=0, fd=10 …)

Dans le cadre de la mise en place d'un service équivalent mais réparti, l'implantation d'un mécanisme d'agrégation réelle est plus complexe à mettre en œuvre (une agrégation réelle engendrant un unique accès transmis au serveur). La figure 6.3 illustre ces propos.

L'application de 4 processus est cette fois-ci répartie sur 4 nœuds distincts et le service de régulation est déployé sur un $5^{\text{ème}}$ nœud. De manière identique lors de l'étape 1, les 4 requêtes sont agrégées et un unique accès est transmis au serveur. La première différence, non la moindre, est la perte d'informations vis à vis des nœuds pour le serveur. Ainsi, lorsqu'une écriture intervient, le serveur qui assure le maintien de la cohérence ne peut aller invalider les

caches des nœuds dont il n'a pas connaissance. C'est donc au service de régulation de réaliser cette tâche.

Connu sous le nom de mémoire partagée distribuée [2] , ce problème a été largement étudié et plusieurs solutions ont été proposées [IS99].
La complexité de ces mécanismes et le coût qu'ils peuvent engendrer sont souvent discutés. Ainsi, nous avons préféré nous appuyer sur une approche intermédiaire qui va nous permettre de nous libérer de cette contrainte de cohérence. L'idée est la suivante : le service de contrôle va définir simplement un ordre entre les requêtes et va informer chacun des clients lorsqu'il devra transmettre sa requête au système de stockage.
Par exemple, les 3 requêtes suivantes read(30,40), read(20,30) et read(10,20) [3] vont être réordonnancées et exécutées dans l'ordre suivant : read(10,20), read(20,30) et read (30,40). De cette manière, les accès sont traités de manière contigüe. Par analogie avec le terme de «requête virtuelle» que nous avons introduit lors de la description de la stratégie d'ordonnancement (cf. section 5.3.1), nous désignerons par la suite, ce processus sous le nom d'agrégation virtuelle.

Cette approche d'agrégation virtuelle nous permet de nous libérer des contraintes de mémoires distribuées en plus de n'avoir aucun impact sur la cohérence du système de stockage. Nous rappelons que la stratégie d'ordonnancement prend en compte les dépendances entre les requêtes de lecture et d'écriture : chaque requête est estampillée lors de son arrivée dans le service de contrôle. Ainsi, lorsqu'une écriture est en attente d'exécution pour un fichier donné, toute lecture sur ce même fichier avec une estampille plus récente ne pourra la précéder.

6.1.3. Synchronisation des requêtes

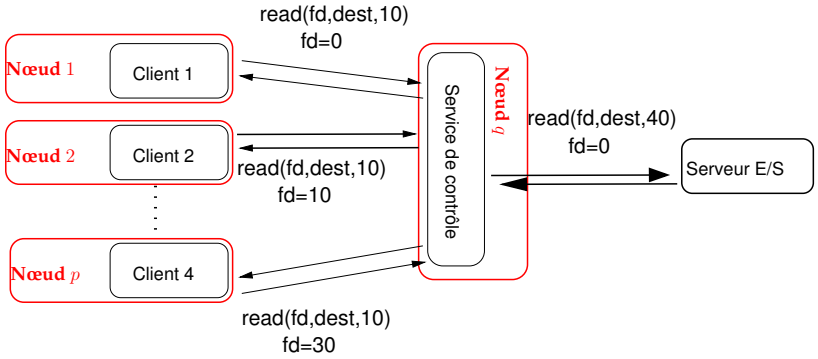
Afin de sérialiser les accès tout en appliquant l'ordre déterminé par la stratégie d'ordonnancement, une des principales difficultés consiste à mettre un mécanisme permettant de réguler les demandes autour du service d'optimisation. Ce mécanisme doit être le moins coûteux possible afin de maximiser l'utilisation du serveur de stockage et limiter l'inactivité du disque entre deux accès.

Cette contrainte peut facilement être ramenée à un problème de type exclusion mutuelle distribuée. En effet, plusieurs clients veulent accéder à la même ressource partagée (le serveur de données). C'est ce problème que nous abordons dans cette partie.

Exclusion mutuelle distribuée

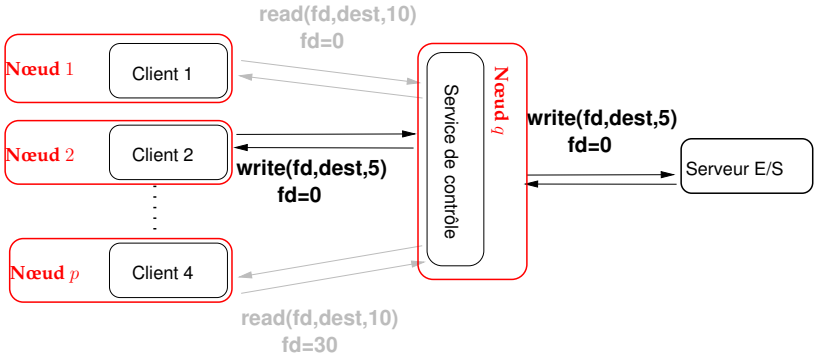
Le problème d'exclusion mutuelle a été étudié depuis fort longtemps, principalement dans les recherches concernant la synchronisation de processus. Plusieurs méthodes permettant de résoudre ce problème sont décrites dans plusieurs manuels de référence [SS94] [Tan01].
Dans un système qui se compose d'une 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.
Afin de justifier la technique qui a été retenue pour réguler les accès au sein de notre proposition, nous rappelons ici plusieurs généralités à propos de ces solutions.

[2] *Distributed Shared Memory*.
[3] `read(x,y)` correspond à une lecture de l'*offset* x à l'*offset* y



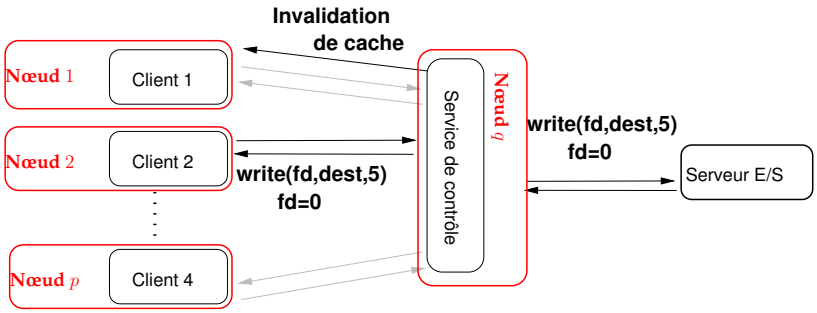
Étape 1

4 accès en lecture sont agrégées



Étape 2

Le nœud 2 réalise une écriture sur une portion préalablement récupérée par le nœud 1.
Les données dans le cache du nœud 1 ne sont alors plus valides.La cohérence n'est pas maintenue.



Étape 2 - bis

Le nœud 2 réalise une écriture sur une portion préalablement récupérée par le nœud 1.
Les données dans le cache du nœud 1 sont invalidées. L'intégrité des données est ainsi conservée.

FIG. 6.3 – Agrégation et cohérence au sein d'un environnement distribué

Algorithmes usuels

Plusieurs algorithmes d'exclusion mutuelle distribuée sont disponibles. Chacune des stratégies permet d'optimiser un ou plusieurs critères, nous donnons ici les plus pertinents pour notre problème :

- Le nombre de messages utilisé dans l'algorithme en fonction du nombre de clients.
- Le délai de synchronisation qui correspond au temps entre l'événement «la ressource est libérée» et l'événement «la ressource est de nouveau occupée» (noté ds). Ce temps nous intéresse particulièrement puisque associé au temps d'accès, il représente la période d'inactivité de la ressource.
- Le temps de réponse, c'est le temps entre l'envoi d'un message de demande d'accès à la ressource et le moment où l'accès a été traité par la ressource. Cette valeur correspond à l'interactivité du système.
  Afin d'éviter les problèmes de famine, la plupart des mécanismes d'exclusion mutuelle satisfont le critère suivant : chaque requête accède à la ressource selon son ordre d'arrivée dans le système. Dans notre cas, cette métrique n'est pas appropriée puisqu'elle est dépendante du critère d'interactivité fourni par notre stratégie d'ordonnancement.

Trois approches sont principalement utilisées pour mettre en œuvre l'exclusion distribuée :

- Les modèles client/serveur :
  Une manière simple consiste à centraliser les demandes autour d'un site de contrôle. Chaque fois qu'un site veut accéder à la ressource, il envoie un message au site de contrôle, le site de contrôle garde cette requête dans une queue et informe le site quand la ressource se libère. Bien que cette méthode soit assez simple à implanter, elle comprend deux défauts majeurs. D'une part, les inconvénients propres à un point central, à savoir le passage à l'échelle et la tolérance aux pannes et, d'autre part, le fait que le délai de synchronisation est dépendant de la latence de l'architecture sous-jacente ($ds \geq 2T$ avec T correspondant au temps nécessaire de transmission d'un message entre deux clients).
- Les algorithmes à jeton :
  Dans ce type d'algorithme, un jeton unique est partagé entre tous les clients. Il est nécessaire de possèder le jeton pour accèder à la ressource. Dans ce type de stratégie, la principale mission est de gérer le mouvement du jeton. L'algorithme le plus simple consiste à 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 ($ds \geq T * (n-1)$ dans le pire cas avec n le nombre de clients). Plusieurs variantes ont été proposées [SK85] [Ray89] afin de diminuer le nombre de messages et le temps de synchronisation.
- Les algorithmes de type innondation :
  Ces algorithmes reposent sur l'idée suivante : 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 [Lam78]. Cet algorithme requiert un délai de synchronisation au mieux de $ds \geq T$ et nécessite $3(n-1)$ messages à chaque itération. Plusieurs optimisations ont été suggérées [RA81] [SS94] mais ils utilisent tous un grand nombre de messages et sont donc comme la première classe fortement dépendant des performances réseaux.

Algorithme retenu

Afin de faciliter la mise en œuvre et l'évaluation de notre proposition, nous avons choisi de mettre en place une stratégie simple s'appuyant sur un unique site de contrôle. Cette approche nous permet d'associer au site de contrôle, assurant l'exclusion mutuelle distribuée, la stragégie d'ordonnancement. Nous restons cependant bien conscient qu'une telle solution comporte plusieurs défauts comme le passage à l'échelle ou encore la tolérance aux pannes. Toutefois, elle facilite énormément la mise en place de notre architecture, et donc, l'évaluation des concepts de sérialisation et de réordonnancement global qui est, nous le rappelons, notre principal but.

La figure 6.4 décrit les différentes étapes lorsque deux clients souhaitent accéder à la ressource.

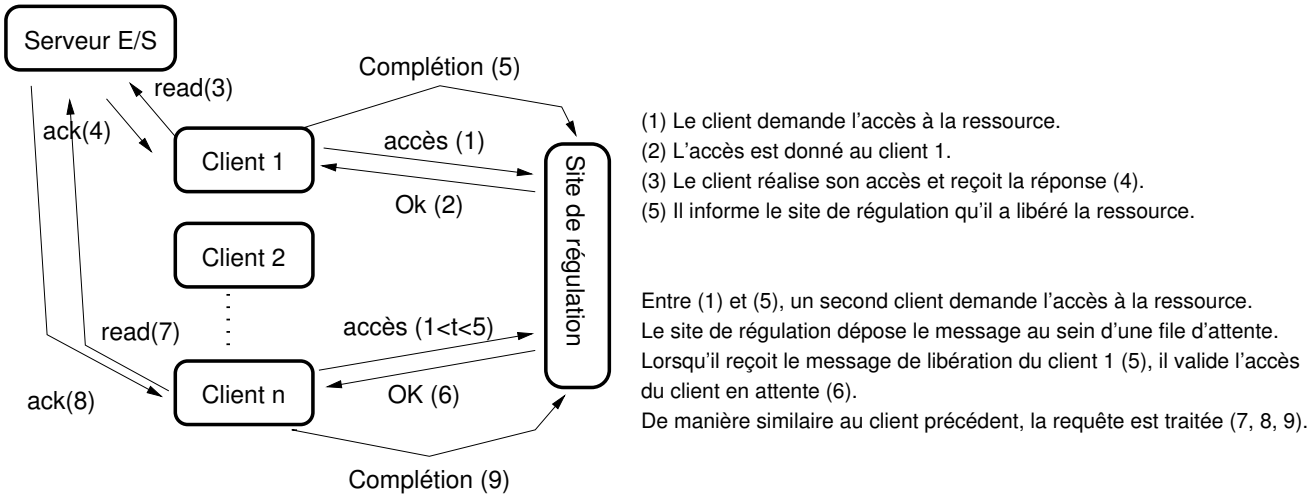


FIG. 6.4 – Synchronisation des accès

A partir de cette architecture logicielle (modèle client/serveur, agrégation virtuelle), nous avons implanté un service en espace utilisateur. Ce premier développement nous a permis de soulever plusieurs problèmes. Nous les abordons par la suite.

6.1.4. Implantation

En s'appuyant sur l'architecture logicielle définie, nous avons développé un service basé sur un modèle client/serveur en langage C. Ce système se décompose en deux parties : la première, liée à l'application, permet de surcharger les appels `POSIX` depuis l'espace utilisateur afin de les transmettre vers le service de régulation. Ce service, multi-threadé, correspond à la seconde partie de notre système. Les requêtes clientes sont transmises par un canal `tcp` au serveur de régulation. Un *thread* réceptionne les diverses requêtes et les trie dans les différentes queues selon le fichier qu'elle concerne, l'*offset* d'accès, leur taille et enfin leur heure d'arrivée.

(cf. section 5.3.1). Par la suite, un *thread* par système d'E/S à contrôler, est utilisé afin de transmettre les messages d'autorisation directement au client dans le cas d'un modèle à plat (un seul site de contrôle) ou à destination d'un niveau supérieur préalablement défini si une structure hiérarchique est exploitée.

Ce prototype, intitulé *aIOLimaster*, nous a permis de réaliser une première série de tests autour d'une application parallélisée fortement dépendante des E/S. L'application parallèle était déployée sur 16 nœuds et utilisait un unique site de contrôle sérialisant et optimisant les accès.
Deux configurations ont été évaluées (figure 6.5) : le service est d'abord déployé sur un nœud quelconque de l'architecture, puis sur le serveur de stockage, afin d'étudier l'impact d'un tel module sur le *CPU* et le réseau et donc sur le serveur `NFS` lui même.

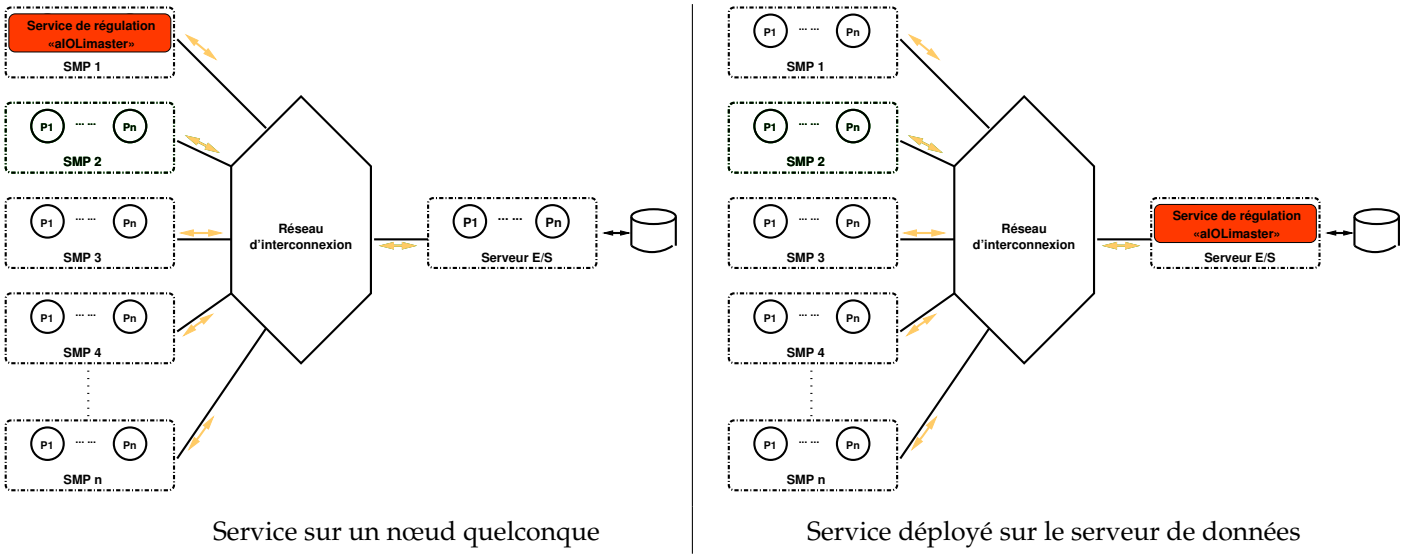


FIG. 6.5 – Vers une solution non-intrusive - *aIOLimaster*

Les résultats obtenus [LDV05] nous ont permis de déceler plusieurs problèmes inhérents à l'architecture logicielle mise en œuvre, nous les abordons ci-après.

- La surcharge des appels en espace utilisateur retarde l'utilisation du cache :
  Chaque requête est transmise au serveur de régulation *aIOLimaster* sans vérifier si la requête peut être servie par le cache client. La requête est ainsi temporisée inutilement. D'une manière générale, si le mode d'accès est séquentiel et que la granularité d'accès est inférieure à la granularité du système de fichiers, le système *aIOLimaster* ajoute un surcoût significatif. Ce phénomène s'annule lorsque la granularité devient importante et l'apport de la sérialisation devient alors visible. Cependant, le gain reste largement limité par le second phénomène.
- Une sous exploitation du système de stockage entre deux accès.
  Entre la fin d'un accès (étape 4, figure 6.4) et la réception de la nouvelle requête (fin de l'étape 7), le système de stockage est inutilisé. Lorsque le nombre d'accès devient élevé, le temps «perdu» devient significatif et le surcoût associé non négligeable.
  A titre indicatif, une décomposition d'un fichier de 2 Go à une granularité de 8 ko engendre 256000 requêtes. Ainsi, avec une latence moyenne de $T = 50\mu$s [4] et une période

[4] Latence mesurée sur notre plate-forme expérimentale, cf. section 4.1.1.

d'inactivité d'au moins $4T$ imposée par le modèle de synchronisation implanté, le système de stockage est inactif pendant plus de 50 secondes. Or dans notre cas, les disques utilisés proposant un débit aux alentours de 57 Mo/s, 36 secondes devrait être suffisantes pour récupérer cette taille. Nous avons donc tenté d'intégrer une approche à base de prédiction permettant de recouvrir ces délais. Elle est décrite au prochain paragraphe.

6.1.5. Régulation à base de prédiction

L'implantation d'un prototype basé sur une coordination autour d'un site maître nous a permis d'identifier l'importance du délai de synchronisation et de la période d'inactivité qui en résulte. Afin de diminuer ce délai si possible à zéro (dans le cas optimum), nous avons choisi de nous appuyer sur une méthode à base de «prédiction» du temps d'exécution de la requête en cours. La nouvelle méthode de synchronisation, illustrée par la figure 6.6, se déroule de la manière suivante :

Côté client :

- Chaque fois qu'un client souhaite accéder au serveur de stockage, il transmet un message de `DEMANDE` d'accès au site de contrôle (dans notre cas *aIOLimaster*) et attend jusqu'au moment où il reçoit un réponse `PREPARATION` contenant un délai préalablement calculé.
- Le client active une alarme initialisée sur le temps fourni par le site de régulation. La requête sera réalisée lorsque le temps d'attente se sera écoulé.
- Lorsque la requête a été satisfaite, le client envoie un message de `FIN` au site de contrôle.

Côté site de contrôle / *aIOLimaster* :

- Quand le site de contrôle reçoit un message `DEMANDE`, il insère comme précédemment la requête dans la file d'attente adéquate afin d'appliquer la stratégie d'ordonnancement.
- Quand le site de contrôle reçoit un message `FIN`, si une requête était en attente de la ressource, il sait qu'elle a commencé à exécuter son opération. Il va estimer le temps de transfert de ce client et puis envoyer un message `PREPARATION` au client dont la requête vient d'être sélectionnée par l'algorithme d'ordonnancement. Le temps de transfert est estimé par la formule suivante :

$$\textit{temps de transfert} = \frac{\textit{taille de requête}}{\textit{debit\_estimé du disque}} \tag{6.1}$$

D'un point de vue théorique, aucune période d'inactivité apparaît et le nombre de messages pour chaque tour de synchronisation est de 3 (DEMANDE, PREPARATION et FIN). Malheureusement, le succès de cette méthode est lié, en plus des caractéristiques réseau, à la justesse de la prédiction du temps de transfert des données.
Dans la pratique, trois cas sont possibles :

- Dans le cas optimal, le temps réel est égal au temps prévu, figure 6.6 (a).
- Dans le cas le plus défavorable, le temps réel est supérieur au temps prévu et le second client transmet sa demande trop tôt, figure 6.6 (b). Les accès se recouvrent et les performances sont dégradées.
- Le dernier cas survient lorsque le temps réel est inférieur au temps prévu ; les capacités du disque sont alors sous-exploitées, figure 6.6 (c).

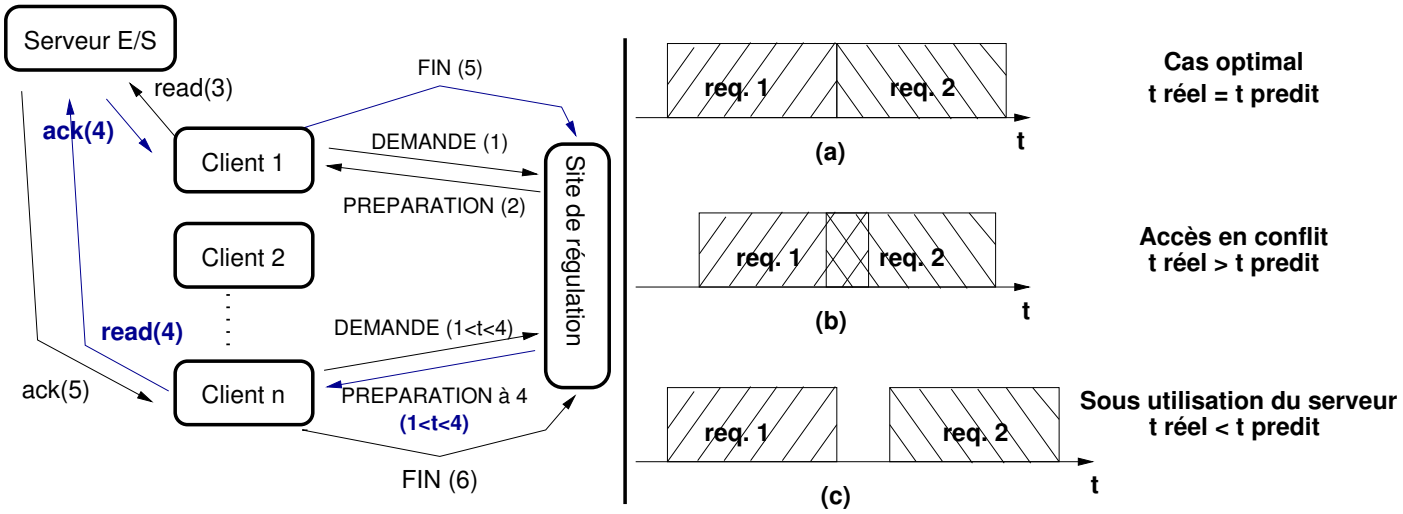


FIG. 6.6 – Synchronisation des accès
A gauche, la figure décrit les différentes étapes de synchronisation. A droite, l'utilisation du disque est présentée selon la précision de la prédiction.

Pour résoudre le problème de recouvrement entre les accès qu'il est impératif d'éviter, nous pondérons le temps de transfert prévu par un ratio k (fixé selon les caractéristiques matérielles). Ainsi, le temps réel est toujours inférieur ou égal au temps prévu. Cependant, cette correction peut mener à la situation où le délai de synchronisation est supérieur à $2T$. Ainsi à chaque fois que le site de régulation reçoit une requête FIN, il compare le temps d'exécution réel avec celui prévu. Si l'écart est supérieur à T ($t_{réel} - t_{prévu} > T$), il transmet immédiatement un message `SUPPLEMENTAIRE` permettant de notifier au client en attente qu'il doit commencer son transfert au plus tôt. L'avantage de cette approche est que la période d'inactivité du disque est bornée à $4T$ et le nombre de messages maximum pour chaque synchronisation est de 4 (`DEMANDE`, `PREPARATION`, `SUPPLEMENTAIRE`, `FINI`)

Même si cette approche a permis de fournir de meilleurs résulats, les performances obtenues n'ont pas correspondu à nos attentes [LDV05]. Lorsqu'une prédiction s'avère erronée, elle entraîne une dérive récurrente plus ou moins importante sur l'ensemble des prédictions suivantes. En effet, si la durée d'une opération est déterminée de manière fausse, le message PREPARATION de la requête suivante est également erroné. La nouvelle requête va commencer trop tôt engendrant des turpitudes entre les deux accès et donc une dégradation des performances. De manière similaire, la seconde prédiction sera également fausse suite à la diminution des performances, et va engendrer une nouvelle dérive sur la suivante, et ainsi de suite. La différence entre le temps prédit et le temps réel va par conséquent grandir à chaque nouvelle erreur. Le système a été légèrement amélioré afin de prendre en compte la dérive présente à chaque nouvelle prédiction. Cependant, là encore, les résultats n'ont pas été convaincants. En effet, il semble compliqué de fournir une prédiction fiable pour des accès portant sur une faible quantité de données [SB02].

6.1.6. Bilan

Nous avons décrit une implantation transparente et non intrusive d'un service de sérialisation et d'ordonnancement global à une grappe. Les principes, les contraintes et les solutions retenues pour la mise en œuvre d'un tel service ont été abordés. Les expériences conduites sur ce modèle n'ont pas été présentées. En effet, même si les résultats obtenus [LDV05] ont montré plusieurs gains susceptibles d'être apportés, par une stratégie d'ordonnancement de plus haut niveau, ils n'ont pas été concluants. Les principales difficultés se sont situées autour de la synchronisation des requêtes.

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 complexe à mettre en œuvre. La difficulté d'une prédiction rigoureuse pour les accès de faible taille en est la principale cause. De plus, chaque prédiction erronée entraîne une dégradation progressive sur l'ensemble des prédictions suivantes. Le développement en parallèle d'une approche plus intrusive en mode noyau nous a amené à laisser temporairement cette première approche. Nous décrivons cette seconde implantation dans la section suivante.

6.2. *aIOLi*, un support pour ordonnancement haut niveau

Certains concepts décrits lors de la section précédente présentent plusieurs limitations ne permettant pas une mise en œuvre efficace d'une stratégie d'ordonnancement globale, principalement la centralisation explicite des requêtes autour d'un site de régulation. Cette seconde proposition consiste à exploiter des points centraux logiciels déjà présents dans l'architecture afin d'y plaçer l'algorithme d'ordonnancement global. De cette manière, la demande émise par un client ne transitera pas par un point supplémentaire : l'aller/retour entre le client et le site de régulation, requis avant d'émettre réellement la demande vers le serveur, étant supprimé. La stratégie d'ordonnancement sera appliquée à un endroit précis entre le client et le système de fichiers. Cette solution est plus intrusive d'un point de vue système mais offre une transparence complète vis à vis des applications. En effet, le service d'ordonnancement est transféré depuis les clients (les applications) vers les systèmes d'E/S.

Ce nouveau positionnement au sein de la pile logicielle de la gestion des E/S permet de collecter un maximum d'informations en plus d'éliminer les contraintes de synchronisation présentes dans une solution applicative. Nous allons aborder plus en détails ces aspects.

6.2.1. Présentation générale

Cette seconde proposition a été développée dans le but de fournir un service proposant diverses stratégies d'ordonnancement d'E/S d'une manière la plus générique possible. Cette caractéristique a été définie afin de permettre à notre solution d'être employée à différents endroits dans la pile logicielle et sur un maximum de support de stockage.

L'architecture interne du système reste très similaire à celle introduite précédemment (figure 6.7) : les requêtes sont déposées dans des queues et ordonnancées par différents *threads*. Le principal changement intervient au niveau des clients qui ne sont plus les applications mais les systèmes d'E/S. Nous insistons bien sur ce concept quelque peu inhabituel afin de faciliter la compréhension du lecteur. Ainsi, par la suite le terme «client» désignera uniquement un système d'E/S.

Dans cette nouvelle architecture, nommée *aIOLi*, un client peut être soit la totalité du sous-système des E/S d'un noyau, soit un système de fichiers ou encore un service d'E/S plus spécifique. Lorsqu'un client souhaite utiliser le service *aIOLi*, il appelle une routine d'initialisation lui permettant d'être associé à un contrôleur d'E/S et de sélectionner une stratégie d'ordonnancement.

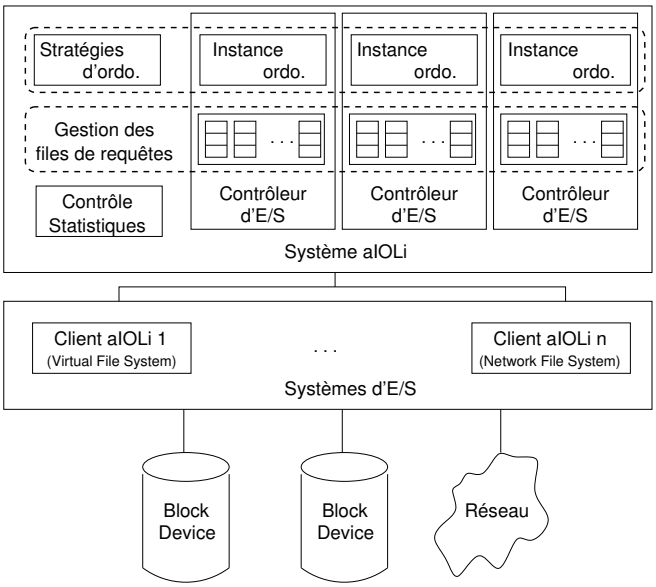


FIG. 6.7 – Architecture du système*aIOLi*
Les clients (les systèmes d'E/S) postent les requêtes au sein des différentes queues et sont informés par *aIOLi* lorsque ces dernières doivent être traitées.

Par la suite, le client est en attente d'un des deux événements suivants :

- Une nouvelle requête arrive au sein du système de fichiers. Ce dernier la dépose alors dans la queue adéquate du contrôleur qui lui a été associé. Les critères de tri restent les mêmes : descripteur de fichier, taille de la requête et *offset* de départ.
- Le module de contrôle applique la stratégie d'ordonnancement sélectionnée et notifie au client (le système de fichiers) qu'une ou un ensemble de requêtes doit être traité. Le client exécute alors la tâche. Comme précédement (cf. section 6.1.2), nous avons choisi de laisser le traitement effectif de la requête à la charge du client. Lorsque des requêtes sont agrégées par le contrôleur d'E/S, il les délivre de manière sérialisée au client. Cette propriété permet à notre système d'être indépendant des clients, les contraintes du maintien de la cohérence (gestion des réplicas, invalidation des caches, ...) restant à la charge du système de fichiers.

  Ce choix se justifie d'autant plus puisque nous souhaitons que le système *aIOLi* soit le plus générique possible de manière à faciliter la connexion à un maximum de systèmes de gestion d'E/S. Or, peu d'entre eux fournissent des routines permettant d'accéder à plusieurs régions en une unique requête. Dans le cas où de telles routines sont disponibles, il est possible d'enregistrer ces fonctions lors de la connexion avec le système de fichiers. Ainsi, les modules de contrôle pourront le cas échéant exploiter ces routines et être encore plus performants.

Nous présentons par la suite chacun des composants de cette nouvelle architecture de manière indépendante.

6.2.2. Notes techniques

D'un point de vue implantation, quatre sous-modules composent ce nouveau prototype *aIOLi*.

Gestion des requêtes entrantes

Cette sous-partie correspond à la structure mise en place afin de stocker temporairement l'ensemble des requêtes en attente de traitement. Elle se compose de deux principales structures :

- Une liste ordonnée selon les estampilles d'arrivées ; chacune des requêtes est identifiée par une estampille logique (un index incrémenté à chaque valeur) et une estampille basée sur la valeur *jiffies* [5] au moment de son arrivée dans le système. Cette liste est utilisée par les algorithmes d'ordonnancement afin d'éviter les problèmes de famine ou encore pour maintenir la cohérence entre les accès de types lecture et écriture.
- Une table de hachage ; elle est utilisée afin d'accéder de manière efficace à toutes les requêtes pour un fichier donné. Chaque fichier est décrit par une structure de données qui contient d'une part deux listes permettant de trier les accès selon leur type (lecture ou écriture) et plusieurs champs fournissant des informations sur les derniers accès traités pour ce fichier. Ces renseignements améliorent le processus de décision des diverses stratégies d'ordonnancement (cf. section 5.3.2).

Lorsqu'un client souhaite déposer une requête au sein du système, il fournit l'identifiant du fichier, le type de la requête, la taille et enfin l'*offset* de départ. Ces renseignements permettent dans un premier temps de déterminer l'emplacement où la globalité de la requête réceptionnée par le client est copiée et, par la suite, d'appliquer la stratégie d'ordonnancement. La structure interne de la requête dépendante du système de fichiers (requête RPC, protocole spécifique, ...) n'est pas exploitée par le système *aIOLi*. L'ensemble de la requête est remise au client lorsqu'elle a été retenue par la stratégie d'ordonnancement.

Stratégies d'ordonnancement

Cette partie est primordiale dans le système *aIOLi* puisqu'elle correspond aux différents algorithmes d'ordonnancement disponibles. Au démarrage du système, chacune des stratégies est instanciée. Lorsqu'un système demande à utiliser le service, il indique la politique d'ordonnancement qu'il désire parmi celles présentes dans le «*pool*» d'ordonnancement.
Cette partie a été implantée de manière à pouvoir ajouter facilement des nouvelles stratégies. Un tableau de taille dynamique référence l'ensemble des codes propres à chacune des politiques. Ainsi, le service *aIOLi* peut être vu comme un support permettant d'évaluer plusieurs stratégies d'ordonnancement. Ce dernier aspect qui n'avait pas été prévu dans un premier temps se révèle être un des points forts de cette proposition.

Les algorithmes disponibles ont accès à l'identifiant des fichiers, à la taille et à l'*offset* de départ de chaque requête ; ces renseignements permettent de fournir un ordonnancement plus fin en comparaison aux stratégies de plus bas niveau basées uniquement sur les secteurs disque. Le module actuel est fourni avec deux stratégies : une politique standard *FIFO* et la variante de

[5] Le nombre de tops d'horloge survenus depuis le démarrage de la machine.

l'algorithme MLF décrite précédemment. L'implantation du système a été réalisée de manière à faciliter l'ajout de stratégies (soit directement au sein du code soit par l'implantation d'un module secondaire). Le développement d'une troisième stratégie [LDV05], intitulée «*Weighted SJF*» est en cours.

Contrôleur d'E/S

C'est l'acteur principal du système, il contrôle, réordonnance et régule les E/S pour un ou plusieurs clients. Le contrôleur élit par le biais de la stratégie d'ordonnancement les requêtes à traiter et ce tant que les files d'attentes de l'ensemble des fichiers qui lui incombent ne soient pas vides. Quand le cas se présente, le contrôleur passe dans un état d'inactivité.

D'une manière plus générale, lorsqu'un client souhaite utiliser le service *aIOLi*, il appelle une routine d'initialisation en lui indiquant l'identifiant de l'unité de stockage à laquelle il est rattaché. Si cet identifiant est déjà pris en charge par un contrôleur, le client va simplement être associé à ce dernier. Dans le cas contraire, un nouveau contrôleur est démarré et initialisé avec la stratégie d'ordonnancement sélectionnée par le client. Cette approche permet à notre système de contrôler et réguler les E/S de manière indépendante ou globale (si cela est applicable) et d'améliorer ainsi les performances. Par exemple, pour un serveur `NFS` qui exporte une partition `ext3`, *aIOLi* peut ordonnancer les requêtes entrantes pour le serveur `NFS` tout en tenant compte des accès locaux réalisés sur la partition `ext3`. Dans ce cas, le serveur `NFS` et le système de fichiers `ext3` seront deux clients associés au même module de contrôle.

Contrôle statistique

La dernière partie a été développée afin d'obtenir des statistiques sur les différentes optimisations apportées par le système. Le fichier `/proc/aioli/stats` permet de visualiser ces informations en temps réel ou de manière *post-mortem*. Pour chacun des fichiers en cours de traitement, il est possible de connaître, par exemple, le nombre de requêtes traitées, le nombre d'agrégations réalisées au total et en moyenne, la taille de la plus grande et de la dernière agrégation, le nombre de décalages/d'attentes détectés et corrigés (cf. section 5.3.2).

Interface d'utilisation

L'interface du système *aIOLi* est relativement simple. Elle se compose de 5 fonctions :

```
struct client {
        void (*process_request)(struct request_t *req);
        void (*process_requests)(struct list_head *reqs);
};
int aioli_init(struct client *clnt,...);
void exit aioli_exit(void);
int aioli_add_request(void *file_id, int type, loff_t offset,
        ssize_t len, internal_data *data, struct client *clnt);
```

La première étape pour chacun des clients consiste à appeler `aioli_init` en passant en argument une structure contenant deux pointeurs vers les fonctions : `process_request` et `process_requests`. Ces fonctions sont utilisées par le contrôleur d'E/S associé afin de redistribuer respectivement une requête ou un agrégat de requêtes au client.

A chaque fois que le client reçoit une nouvelle requête, il la dépose grâce à la fonction `aioli_add_request` en passant en argument l'identifiant du fichier, le type de requête (lecture ou écriture), l'*offset* de départ, la taille. Le pointeur `data` permet de copier la totalité de la requête délivrée au client afin de la lui redonner par la suite en utilisant le pointeur `clnt`.

Enfin, un client est retiré proprement du système par l'appel de `aioli_exit`.

Cette interface est relativement réduite et donc simple. Cependant l'utilisation du service requiert une étape qui reste très intrusive. En effet, il est nécessaire d'aller instrumenter le code du système d'E/S afin d'intercepter les requêtes entrantes et les rediriger vers le service d'ordonnancement. Cette étape requiert donc une bonne connaissance de l'architecture logicielle du système de fichiers qui souhaite être interconnecté au service d'ordonnancement.

Le module *aIOLi* actuel est fourni avec les codes nécessaires permettant de modifier le système de fichiers `NFS` ainsi que la couche d'abstraction `VFS`. Nous présentons brièvement l'implantation de ces deux clients.

6.2.3. *aIOli* et `NFS`

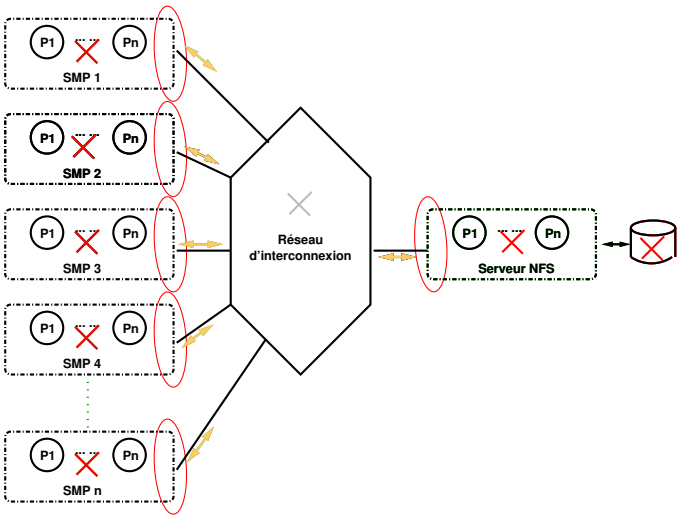
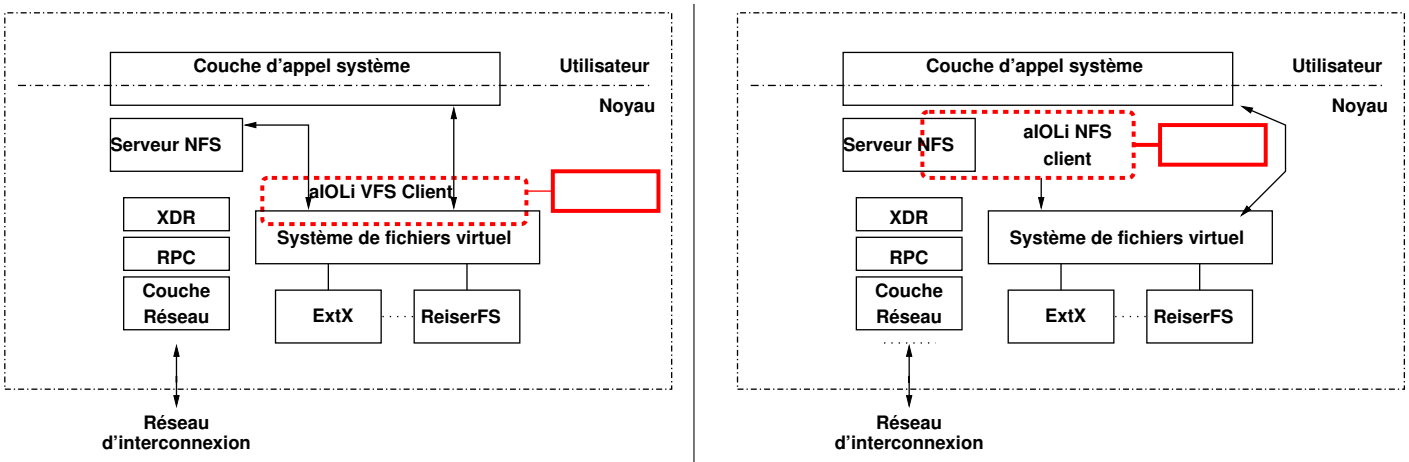


FIG. 6.8 – Points de centralisation propres à une architecture basée sur un serveur centralisé

Sur une architecture proposant un système de fichiers centralisé à la `NFS` (cf. figure 6.8), les points susceptibles de répondre à nos attentes sont les noyaux soit côté client soit côté serveur. Or, comme nous l'avons vu au chapitre 5, la mise en place d'un service au sein de chaque client ne permet pas une régulation globale. Il en est également de même pour des systèmes de fichiers parallélisés. Cependant dans le cadre d'un système `NFS`, la contrainte liée au parallélisme entre les accès est absente et il est envisageable de mettre en œuvre le service au sein du serveur centralisé.

Dans cette section, nous présentons deux possibilités d'interconnexion de l'unité de stockage exportée par le serveur `NFS` : soit au niveau de la `VFS` soit directement au sein du module noyau `NFS`. Les deux solutions sont décrites ci-après.



(a) - Optimisation de la `VFS`

(b) - Optimisation du service `NFS`

FIG. 6.9 – Mise en place du service *aIOLi* sur le serveur de `NFS`
A gauche, l'ensemble des E/S pour un périphérique donné sont traitées par *aIOLi*. A droite, le service est uniquement utilisé par le module `NFS`, les requêtes locales ne sont pas contrôlées par *aIOLi*.

Dans le premier cas, la couche d'abstraction `Virtual File System` est modifiée afin de rediriger l'ensemble des E/S entrantes vers le service *aIOLi* (figure 6.9). Le module *aIOLi* est placé entre l'interface des appels systèmes *Linux* et la *VFS*. Chacune des requêtes qui arrive au niveau de la `VFS` est analysée afin de savoir si elle doit être régulée par *aIOLi* (selon le périphérique indiqué lors de l'initialisation). Si tel est le cas, la requête est déposée dans des queues de notre système en attendant d'être sélectionnée par le contrôleur associé. De la même manière qu'un appel synchrone par la `VFS`, le processus qui a généré l'appel reste bloqué jusqu'au traitement de sa requête.
Cette approche est la plus optimale puisqu'elle contrôle l'ensemble des interactions à traiter sur le serveur : les requêtes arrivant depuis le réseau et celles générées par les processus locaux. Cependant, elle n'est pas la plus appropriée à l'architecture d'un serveur `NFS` basée sur des E/S synchrones. Nous rappelons que le serveur `NFS` comprend un nombre défini de *threads* noyau qui correspond au nombre de requêtes pouvant être traitées en parallèle (4.3.2). De ce fait, en optimisant les E/S au niveau de la `VFS`, les *threads* `nfsd` vont être bloqués jusqu'à la complétion des requêtes, limitant ainsi les améliorations susceptibles d'être apportées avec un plus grand nombre de requêtes.

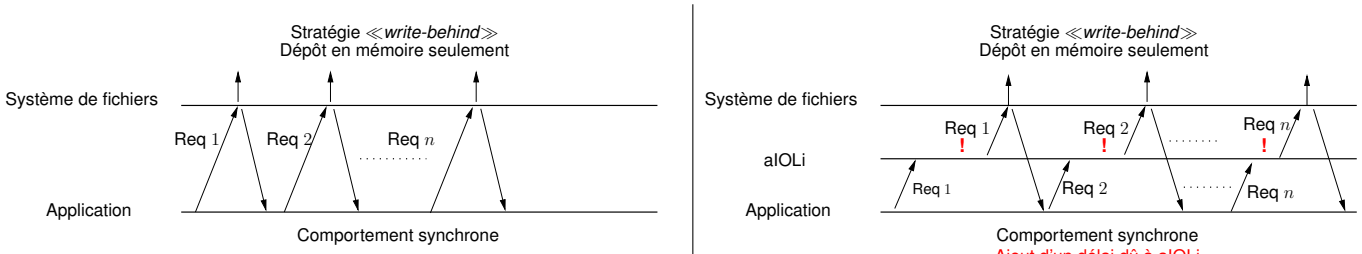
Dans la seconde implantation (cf figure 6.9 (b)), nous avons modifié directement le code du module noyau `nfs` afin qu'il redirige les requêtes vers le service *aIOLi* d'une manière non-bloquante. Le *thread* `nfsd` dépose la requête `RPC` dans notre système puis retourne en attente d'un nouvel événement : une nouvelle requête arrive sur le serveur `NFS` ou le contrôleur *aIOLi* associé invoque un *thread* `nfsd` afin d'exécuter une ou plusieurs requêtes sélectionnées par la stratégie d'ordonnancement. La fonction appelée par *aIOLi* réinjecte les requêtes à l'endroit du code `NFS` où elles avaient été détournées. De cette manière, les *threads* `nfsd` traitent les requêtes et retournent les réponses aux clients. Le service *aIOLi* réalise uniquement un rôle de contrôle, de réordonnancement et de régulation, les autres aspects restant à la charge du service `NFS`.

L'ensemble des tests que nous présenterons dans le chapitre suivant a été réalisé sur la nouvelle architecture logicielle composée de `NFS` et *aIOLi* (cf. chapitre 7).

Avant de conclure, la présentation de cette implantation, nous abordons, une optimisation nécessaire à la gestion de requêtes de type écriture dans le cas où cette dernière est sérialisée.

6.2.4. Gestion des écritures - un cas particulier

La sérialisation permet de bénéficier des stratégies de pré-chargement lors d'accès contigus en lecture : les premières requêtes déclenchent le mécanisme permettant aux requêtes suivantes d'être servies plus rapidement (les données étant pré-chargées dans le cache du système). Lors d'opérations de type écriture, l'apport de la sérialisation est moindre puisqu'il a pour unique but d'assurer un ordre strict permettant de minimiser les déplacements des têtes de lecture/écriture. Néanmoins, les premières évaluations conduites sur le prototype *aIOLi* nous ont permis de déceler un surcoût engendré par les stratégies à base d'écritures retardées présentes au sein des noyaux *Linux* (cf. section 2.2.3). Le schéma 6.10 décrit cette situation.

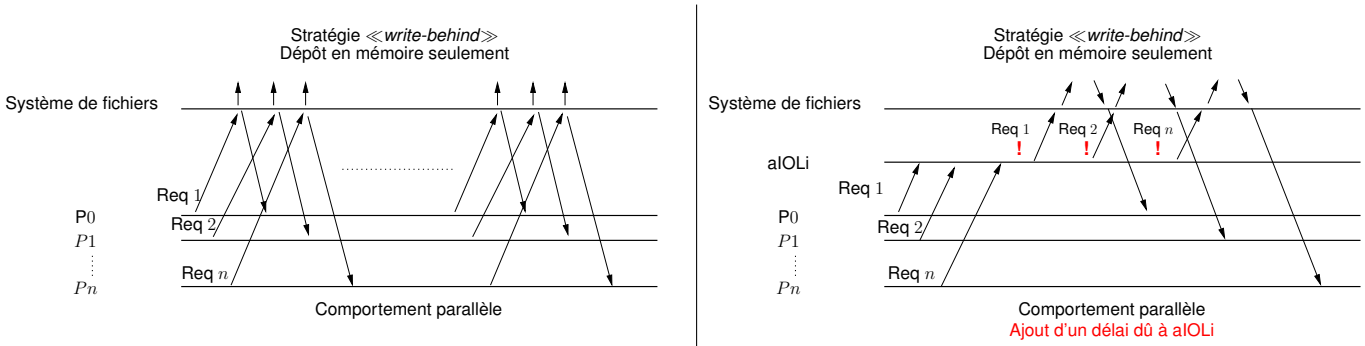


(a) - Système d'E/S standard

(b) - Système d'E/S utilisant *aIOLi*

FIG. 6.10 – Sérisalisation et écriture : cas synchrone

Les stratégies de type écriture retardée permettent aux accès en écriture d'être acquittés plus rapidement : chaque accès est déposé en mémoire (généralement le *buffer-cache*) et la réponse est transmise immédiatement vers le client. Les données ainsi copiées dans la mémoire seront, si nécessaire, transmises vers l'unité de stockage de manière asynchrone. Dans le système *Linux*, le démon `pdflush` effectue cette tâche [Lov05]. Lorsque les écritures sont sérialisées par *aIOLi*, une première copie est effectuée afin de déposer la requête dans les queues associées au contrôleur. Par la suite, le contrôleur sélectionne la requête et la redistribue au serveur. La requête est alors immédiatement acquittée à l'application. Celle-ci peut alors transmettre la demande suivante. Le processus est réitéré durant la totalité de l'opération. De ce fait, le surcoût devient rapidement non négligeable.



(a) - Système d'E/S standard

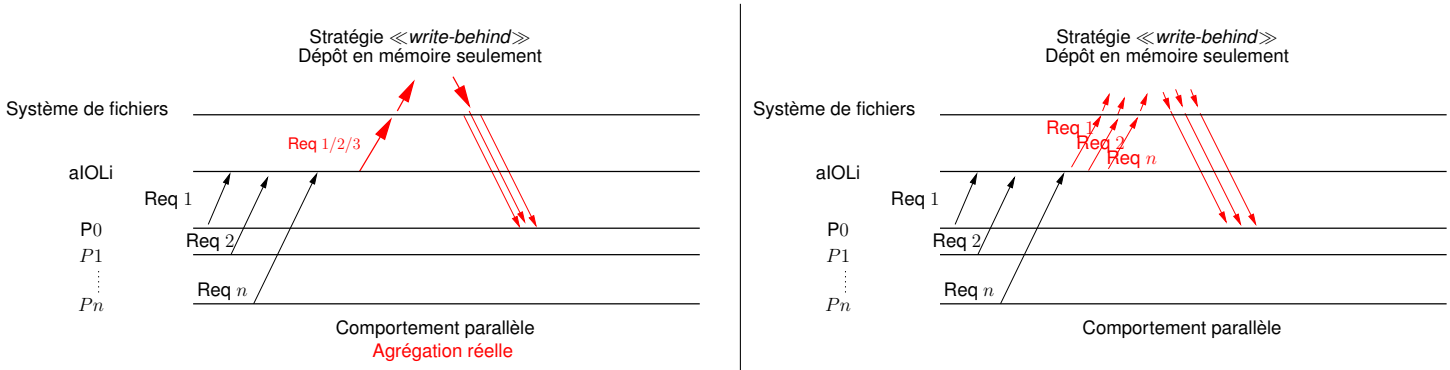
(b) - Système d'E/S utilisant *aIOLi*

FIG. 6.11 – Sérisalisation et écriture : cas parallèle (problème)

Lors de modes d'accès en parallèle, le comportement est identique : les données de chaque accès sont déposées immédiatement dans la mémoire lors de l'utilisation standard du serveur. L'exploitation du service *aIOLi* ajoute un délai pour chacune des requêtes (cf .figure 6.11).

Cependant, si les performances sont meilleures cela n'est que temporaire. En effet, le démon `pdflush` propage les requêtes d'écriture vers le disque selon leur estampille d'arrivée. Les accès arrivant selon un ordre non défini, le fichier est écrit par morceaux non contigus sur le disque. Les performances seront donc dégradées lors de futures lectures.
La stratégie d'ordonnancement mise en place dans le système *aIOLi* permet de corriger ce défaut en transmettant les requêtes pour un même fichier dans un ordre contigu.
En affinant la méthode de sérialisation pour les écritures dans notre système, nous devrions être capable d'atteindre des performances au moins identiques à celles fournies par l'approche traditionnelle et favoriser l'écriture des données de manière contigüe sur le disque.

Deux possibilités peuvent permettre de réduire considérablement les délais lors de comportements parallèles : la mise en place d'une agrégation réelle ou l'utilisation d'une approche basée sur modèle *pipeline*. Elles sont présentées dansla figure figure 6.12 :



(a) - Mise en place d'une agrégation réelle

(b) - Utilisation d'une approche «*pipelinée*»

FIG. 6.12 – Sérialisation et écriture : cas parallèle (solution)

Dans la première approche, les requêtes composants la requête virtuelle sont encapsulées au sein d'un unique accès transmis au serveur. La requête va être exécutée et le serveur a la responsabilité d'envoyer les réponses aux différents processus clients. Aucune des routines présentes au sein du code `NFS` ne permet de réaliser un accès encapsulant plusieurs sous-requêtes. Par ailleurs et d'une manière plus globale, peu de systèmes de fichiers proposent une telle fonctionnalité et puisque nous souhaitons offrir un service le plus générique possible, nous avons opté pour la mise en place d'une sérialisation «*pipelinée*» pour les écritures. Ce mécanisme va permettre de réduire le délai d'attente global pour le traitement de la requête virtuelle tout en favorisant la contiguïté lors de la propagation des écritures. Pour les écritures purement synchrones, le système ajoute un délai qui pour le moment nous semble incompressible.

6.3. Bilan

Nous avons présenté deux prototypes permettant de contrôler, réordonnancer et réguler l'ensemble des E/S d'une grappe. Le premier prototype, nommé *aIOLimaster* repose sur un modèle applicatif client/serveur. Chaque application souhaitant utiliser le service doit être compilée avec la partie cliente du système. L'ensemble des accès est alors redirigé vers un nœud serveur offrant les diverses optimisations.

Le principal défaut de cette approche est le coût induit par la synchronisation entre les différentes requêtes et la période d'inactivité du disque qui en résulte. Nous avons fait remarquer la difficulté de rendre ce temps «transparent». Une approche à base de prédiction a été proposée mais sa mise en œuvre n'a pas été concluante. Il semble à *priori* complexe d'obtenir des prédictions fiables nécessaires pour être efficace.

La seconde approche, qui a été décrite, consiste à exploiter les points centraux déjà présents dans l'architecture logicielle du système de fichiers. Cette approche plus intrusive d'un point de vue système permet d'éliminer en grande partie les délais imposés par la sérialisation. Ce système, intitulé *aIOLi*, a été developpé sous forme d'un module noyau *Linux*. Il fournit aux différents systèmes d'E/S un service d'ordonnancement totalement indépendant et générique. Les applications n'ont en aucun cas connaissance de ce service.

Un des autres points forts de ce système est qu'il a été conçu afin de faciliter l'intégration de nouvelles stratégies d'ordonnancement. Cette fonctionnalité non prévue au départ permet de faire d'*aIOLi* un support d'évaluation d'algorithmes pour la gestion des E/S.

Le système de fichiers `NFS` a été instrumenté de manière à ce qu'il puisse exploiter les services proposés par la solution *aIOLi*. Les évaluations conduites autour de cette architecture logicielle composée de `NFS` et *aIOLi* sont présentées dans le chapitre suivant.

EXPÉRIMENTATIONS 7

La mise en œuvre d'une stratégie d'ordonnancement de l'ensemble des E/S d'une grappe a conclu à l'implantation d'un service en mode noyau. Ce nouveau système, appelé *aIOLi*, offre un service de contrôle, de réordonnancement et de régulation des E/S pour une large gamme de systèmes d'E/S. Une interconnexion entre le système de fichiers NFS et notre proposition a été réalisée.

Dans ce chapitre, nous présentons les expériences menées autour de cette nouvelle architecture de partage de données NFS-*aIOLi*. La stratégie d'ordonnancement activée est la variante *MLF* décrite en section 5.3.1.

Tout d'abord, nous décrirons les évaluations réalisées afin d'étudier le surcoût et l'impact sur le passage à l'échelle lors de l'utilisation du service de sérialisation *aIOLi*. Les jeux de tests Bonnie++ et b_eff_io ont été utilisés.

Par la suite, nous reprendrons l'ensemble des expériences conduites au chapitre 4 afin d'analyser les apports de la stratégie d'ordonnancement sur un serveur NFS. Nous nous intéresserons à l'étude des mécanismes d'agrégation en cadre mono-applicatif en les comparant avec les routines MPI I/O. Nous aborderons ensuite le cadre multi-applicatif qui, nous le rappelons, est notre principal objectif. Comme nous allons le montrer, les performances atteintes sont surprenantes dans les deux cas.

La plate-forme d'expérimentation est la même que celle décrite en section 4.1.1.

7.1. Surcoût et passage à l'échelle

Dans cette partie, nous avons souhaité évaluer les différents surcoûts induits par l'approche sérialisée. Dans un premier temps, nous avons exécuté le jeu de tests Bonnie ++ bien connu

de la thématique des systèmes de fichiers. Dans un second temps, nous présentons les résultats obtenus pour le programme *b_eff_io* permettant d'analyser les performances lors de comportements parallèles. Les performances d'une application composée de 96 instances `MPI` sont analysées.

Pour les deux expérimentations, les options utilisées par les clients pour «monter» la partition exportée par le serveur `NFS` sont : `nfsvers=3,tcp,rw,wsize=32768, rsize=32768, intr, hard,actimeo=0`. La partition `ext3` est montée en `async` sur le serveur (cf. section 4.1).

7.1.1. Le test `Bonnie++`

Bonnie++ [dt] est un programme bien connu pour évaluer les performances d'un système de fichiers. Il permet d'obtenir une bonne estimation des performances pour des opérations d'écriture et de lecture de manière séquentielle puis aléatoire.

Le test d'écriture se décompose en 3 phases :

- *write-char*, écriture d'un fichier octet par octet via la commande `POSIX putc();`
- *write-block*, écriture d'un fichier par bloc de 8 Ko via la commande `POSIX write();`
- *rewrite*, lecture et réécriture immédiate par bloc de 8 ko via les commandes `POSIX read() lseek()` et `write()`.

Les opérations de type lecture sont évaluées de manière similaire en exploitant les commandes `POSIX getc()` (*read-char*) puis `read()` (*read-block*).

Les performances en accès aléatoire sont obtenues en réalisant un total de 8000 déplacements répartis sur la totalité du fichier. Chacun des déplacements est suivi dans 90% des cas d'une lecture et dans 10% des cas d'une écriture par bloc. La valeur retournée correspond au nombre de déplacements qui ont pu être effectués en une seconde.

Un mode supplémentaire permet d'évaluer les performances lors de la création de fichiers. Ce test n'a pas été réalisé, le système *aIOLi* ne traitant que les requêtes sur les données effectives et n'interceptant pas les requêtes sur les «méta-données» (cf. 1.2)

Bien que ce jeu de tests reste une référence pour les systèmes de fichiers locaux, il n'est pas réellement approprié aux environnements de type grappe. En effet, il ne représente pas un comportement réel d'une application parallélisée *HPC* : peu d'applications accèdent aux fichiers octet par octet. La seconde phase du test qui semble plus pertinente (accès séquentielle par 8 Ko), ne correspond qu'à un seul des modes d'accès observés au sein des applications parallèles. Néanmoins, nous avons décidé d'évaluer `Bonnie++` au-dessus d'un serveur `NFS` exploitant le service *aIOLi* afin de jauger l'impact d'une approche sérialisée lors de modes d'accès non prévus par notre stratégie.

Nous avons paramétré `Bonnie++` afin qu'il exécute l'ensemble des tests pour des fichiers de 4 Go (deux fois la taille du cache).

Les valeurs obtenues apparaissent dans le tableau 7.1 et sont également représentées par les histogrammes présentés au sein de la figure 7.1.

Concernant les écritures, un surcoût d'environ 10% apparaît lors de l'écriture séquentielle du fichier octet par octet (*write-char*) : environ 190 secondes sont requises pour réaliser l'opération sur un serveur traditionnel et 210 sur une architecture `NFS-aIOLi`. Le surcoût diminue lorsque la granularité d'accès devient plus importante (*write-block*). Cette diminution s'explique par le délai rajouté par la sérialisation (cf. section 6.2.4). Pour les deux systèmes, l'étape *rewrite* dégrade considérablement les performances.

		Écriture			Lecture		Random
		write-char Mo/s	*write-block* Mo/s	*rewrite-block* Mo/s	*read-char* Mo/s	*read-block* Mo/s	seeks /s
Cas 1	NFS	21.69	28.40	2.00	34.84	43.54	144
1 instance	aIOLi	19.80	28.29	2.03	37.45	48.81	116
Cas 2	NFS	8.26	8.78	1.68	3.55	3.02	28
4 instances	*aIOLi*	7.41	9.59	1.68	4.74	13.39	43

TAB. 7.1 – Évaluation du jeu de tests `Bonnie ++`

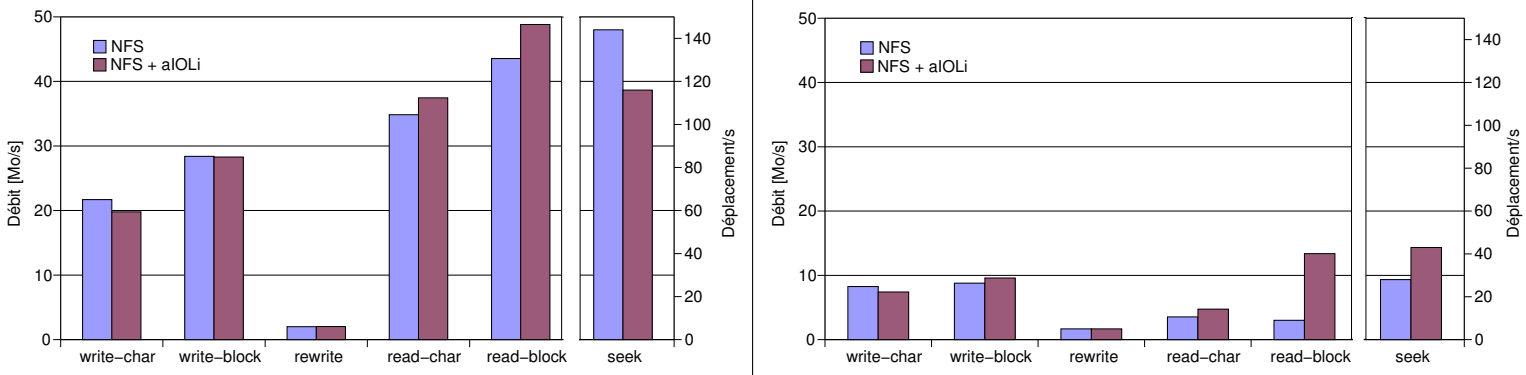


FIG. 7.1 – Évaluation du jeu de tests `Bonnie ++`

Dans le cas d'une utilisation plus intensive du serveur (4 clients `Bonnie++`), l'impact pour les accès octet par octet reste à peu près identique. Cependant lorsque la granularité augmente la stratégie d'ordonnancement global fournie par *aIOLi* permet de mieux répartir la ressource entre les clients et un léger apport apparaît. L'étape de réécriture reste là encore très pénalisante pour les deux architectures.

Concernant les lectures, l'utilisation d'*aIOLi* augmente les performances d'environ 10%. Pour 4 clients, les performances sont multipliées par un facteur 4 lorsque chacun des tests accède aux données par blocs. Comme nous l'espérions, les fenêtres de régulation mises en place par la stratégie d'ordonnancement permettent d'améliorer l'utilisation des mécanismes de pré-chargement (cf. section 5.3.2.2).

Enfin, pour un mode d'accès purement aléatoire (*seeks*), la variante de l'algorithme *Multiple Level Feedback* n'ayant pas été conçue pour de tels comportements, le service *aIOLi* diminue les performances d'environ 20%. Dans le cas multi-applicatif, le découpage par fenêtre permet de réduire les déplacements intempestifs et offre donc de meilleures performances.

D'un point de vue général, le système `NFS` combiné au service *aIOLi* engendre un léger surcoût pour les écritures lors de comportements synchrones mais permet d'améliorer les accès en lecture. De plus, lorsque plusieurs clients indépendants sont lancés en parallèle, la mise en place de fenêtres dédiées pour chacune des applications se révèle être bénéfique.

Dans le paragraphe suivant, nous allons analysé l'influence de la solution *aIOLi* sur le passage à l'échelle.

7.1.2. Le test `b_eff_io`

Pour tester l'impact d'un service sérialisant les accès dans un système de fichiers, nous avons utilisé le jeu de tests *Effective I/O Bandwith* [RKPH01] (version 2.1). Ce programme permet d'évaluer, dans un temps court, différents modes d'accès parallèles en écriture et en lecture (indépendant, collectif, aligné, non-aligné). Il exploite plusieurs routines de la bibliothèque *ROMIO* afin d'évaluer les différentes optimisations proposées par le standard `MPI I/O` (cf. section 3.2.6.1).

Les modes d'accès évalués sont présentés au sein du schéma 7.2 :

- type 0, accès collectif recouvrant (algorithme *Two Phase*) ;
- type 1, accès collectif pas à pas, un accès pour chaque sous-bloc ;
- type 2, accès synchrone sur des fichiers indépendants (1 fichier par processus) ;
- type 3, identique au type 2 mais les fichiers sont regroupés en un seul ;
- type 4, identique au type 3 mais les accès sont faits de manière collective pas à pas.

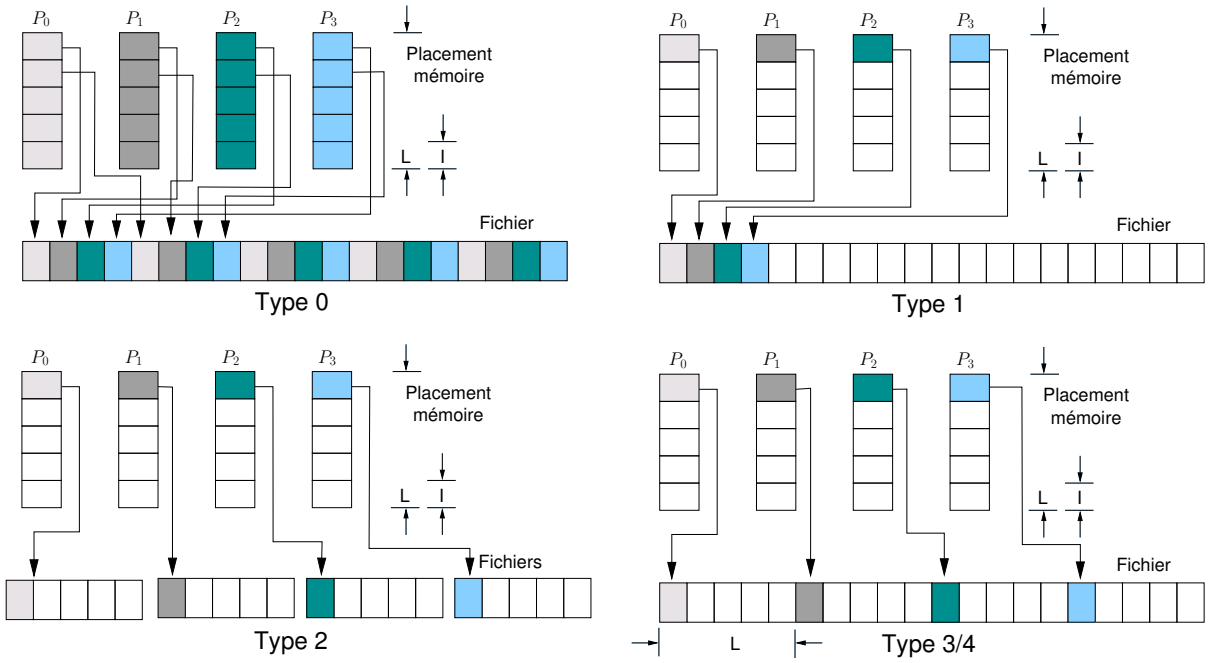


FIG. 7.2 – Modes d'accès utilisés dans `b_eff_io`
Chacun des schémas illustre les données manipulées par un seul appel `MPI I/O`.

Le programme a été lancé sur 2, 4, 8, 16, 32, 64 et 96 instances *MPI*. Chacun des processus *MPI* est déployé sur un unique nœud. Le jeu de tests a été exécuté avec les paramètres par défaut. Chacun des graphiques (cf. figure 7.3 et figure 7.4) est retourné directement par l'application.

Une contrainte de temps d'exécution étant imposée dans le programme, seules les courbes pour le premier mode d'accès en réécriture nous ont été retournées. Les performances sont comparables avec une meilleure régularité pour l'architecture basée sur `NFS`-*aIOLi*. Ce mode d'accès en réécriture n'étant pas détaillé dans la description du jeu de tests, il nous semble difficile de pouvoir effectuer des observations pertinentes sur ce léger avantage. Nous avons choisi de ne pas présenter ces valeurs.

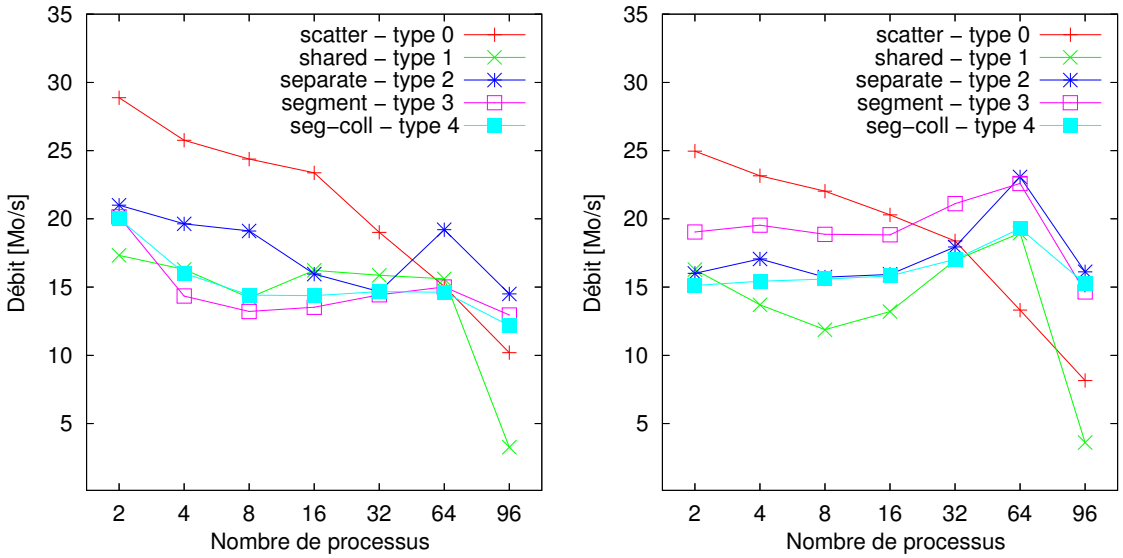


FIG. 7.3 – Analyse de l'impact sur le passage l'échelle en écriture
Comparaison entre un serveur NFS traditionnel (à gauche) et un serveur NFS-*aIOLi* (à droite)

En ce qui concerne les écritures (cf. figure 7.3), les courbes sont très similaires et la sérialisation ne semble pas avoir d'influence sur le passage à l'échelle.

Le premier mode d'accès «type 0» est plus performant sur un serveur traditionnel NFS que sur un serveur NFS-*aIOLi*. En effet, les optimisations sont réalisées par la bibliothèque *ROMIO* qui applique la stratégie collective *Two Phase*. Ainsi, un surcoût est induit par le service *aIOLi* qui souhaite également appliquer sa stratégie d'optimisation. Cependant, à partir d'un nombre important de processus, nous pouvons constater que les mécanismes mis en œuvre par *ROMIO* sont coûteux et ne répondent pas au facteur du passage à l'échelle.

Le second mode, «type 1», correspondant à un comportement séquentiel, n'est également pas bénéfique au service de régulation qui rajoute un délai lorsqu'une politique d'écriture retardée est en place sur le système de fichiers.

Les modes «type 2», «type 3» et «type 4» ne profitent pas des optimisations proposées par MPI I/O qui sont, soit limitées soit inappropriées. Le mode d'accès type 2 qui est comparable à un comportement multi-applicatif n'est, par exemple, pas prévu par le standard. Pour ces comportements, la stratégie proposée par *aIOLi* est efficace et la mise en œuvre des fenêtres de régulation améliore les performances.

Pour les accès en lecture (cf. figure 7.4), le système composé de NFS et *aIOLi* améliore globalement les performances. Au contraire des écritures parallèles qui profitent des stratégies d'écriture retardée présentes sur le serveur, le réordonnancement et la sérialisation des requêtes permettent d'atteindre de meilleures performances pour les lectures.

Pour les algorithmes collectifs mis en place par *ROMIO* pour les modes d'accès «type 0», «type 1» et «type 4», plus le nombre de processus participant est important, plus les performances sont mauvaises. Le service *aIOLi* qui régule les accès permet de limiter ce phénomène en favorisant les techniques de pré-chargement. Cependant, comme pour les écritures, les stratégies de *ROMIO* sont appliquées avant celles fournies dans *aIOLi*. De ce fait, le coût de mise en œuvre d'une politique inappropriée limite les performances susceptibles d'être atteintes par la seule utilisation de notre système. A titre d'exemple, le mécanisme de synchronisation em-

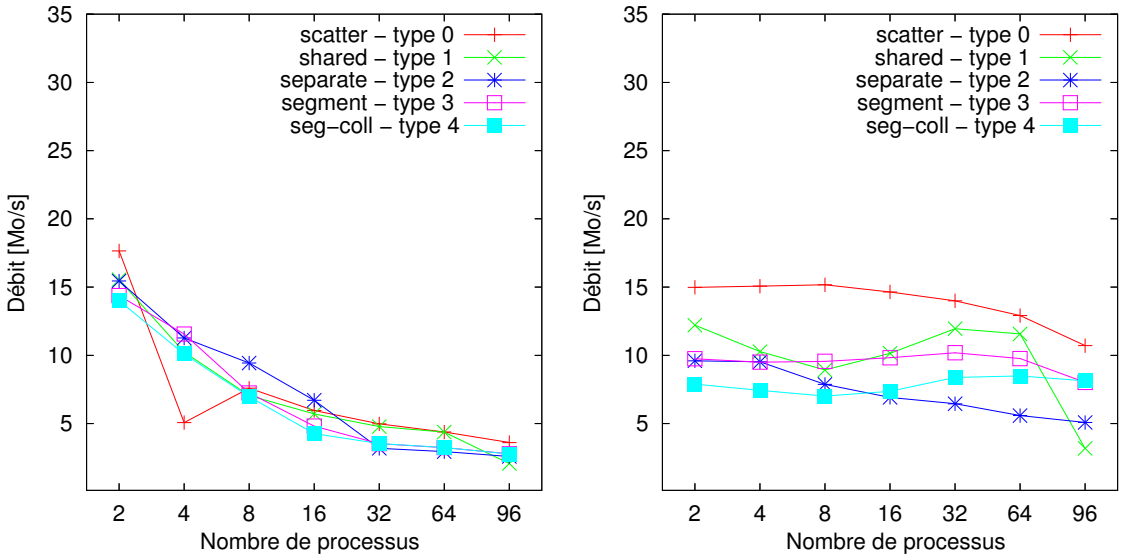


FIG. 7.4 – Analyse de l'impact sur le passage l'échelle en lecture
Comparaison entre un serveur NFS traditionnel (à gauche) et un serveur NFS-*aIOLi* (à droite)

ployé par *ROMIO* pour le mode d'accès «type 1» a un surcoût considérable dès que le nombre de processus devient grand.

Cette expérience nous a permis de montrer que notre système ne diminue pas le facteur de passage à l'échelle, aussi bien en écriture qu'en lecture. Au contraire, il améliore les performances dans plusieurs cas.

Parallèlement, nous avons pu voir que les optimisations prévues par le standard MPI I/O ne sont pas appropriées pour certains modes d'accès. De plus, les stratégies collectives qui améliorent les performances pour un nombre restreint de processus deviennent rapidement coûteuses dès que ce dernier croît.

Le jeu de tests b_eff_io repose uniquement sur l'interface MPI I/O. Or nous souhaitons évaluer notre proposition avec les routines POSIX généralement plus simples : tout d'abord, pour répondre à un de nos objectifs qui consistait à proposer une solution basée sur cette interface mais également, pour savoir si la combinaison de *ROMIO* et *aIOLi* est bénéfique ou non. Nous abordons ces deux aspects dans les prochaines sections.

7.2. Évaluation mono-applicative

Dans cette section, nous reprenons l'ensemble des tests qui a été décrit lors du chapitre 4 et nous les comparons avec l'architecture NFS-*aIOLi*. Nous allons vérifier que la variante de l'algorithme *MLF* permet bien de détecter les comportements parallèles et donc d'agréger un maximum de requêtes. Par la suite, nous évaluerons l'impact du service d'ordonnancement positionné à plus haut niveau sur les ordonnanceurs disque situés en bas de la pile des E/S.

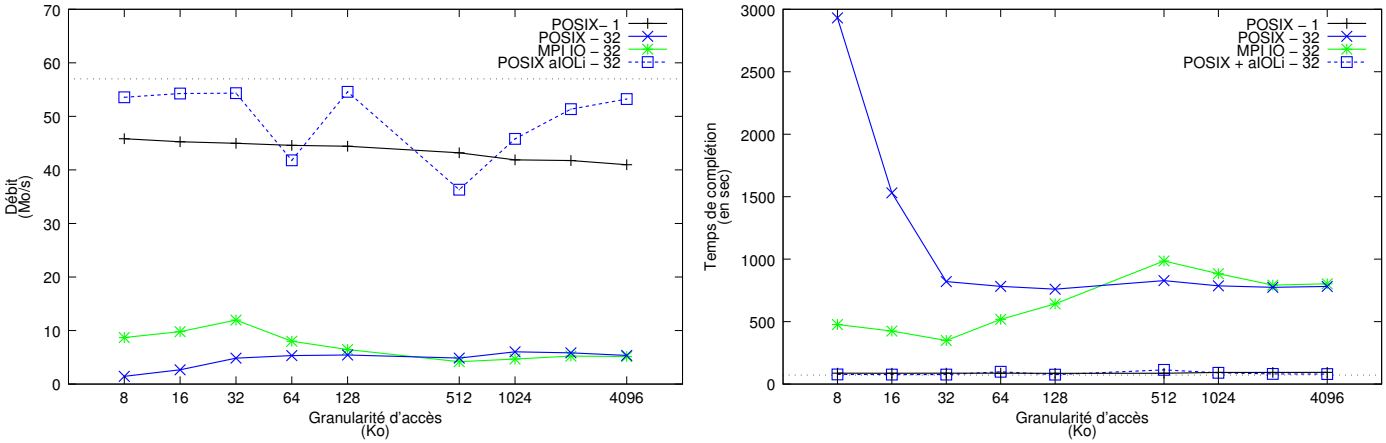
7.2.1. Décomposition : `POSIX`, `MPI I/O`, *aIOLi*

Dans ce paragraphe, nous comparons les performances lors d'une décomposition en lecture puis en écriture entre un serveur `NFS` traditionnel et un serveur `NFS`-*aIOLi*. Les interfaces `POSIX` et `MPI I/O` sont évaluées au-dessus des deux architectures.

Lecture

Une décomposition d'un fichier de 4 Go incluant 32 processus déployés sur 32 nœuds a été réalisée.

Les graphiques de la figure 7.5 permettent de comparer les performances atteintes par les deux systèmes. Les courbes en trait plein correspondent aux mesures obtenues précédemment alors que celles en trait pointillé sont les valeurs nouvellement recueillies.



(a) - Débit (b) - Temps de complétion

FIG. 7.5 – Décomposition d'un fichier de 4 Go sur 32 processus - Lecture - `NFS` *vs* `NFS`-*aIOLi*
Comparaison des performances obtenues par l'utilisation de l'interface `POSIX`, `MPI IO` et `POSIX` au dessus d'un serveur `NFS`-*aIOLi*.

Grâce aux optimisations apportées à la variante de la stratégie *MultiLevel Feedback* (cf. section 5.3.2), l'utilisation de l'interface `POSIX` au dessus de l'architecture `NFS`-*aIOLi* fournit les meilleures performances (courbe «`POSIX aIOLi - 32`»). En comparaison avec les performances fournies par les interfaces `POSIX` et `MPI I/O` sur un serveur traditionnel, le facteur d'amélioration est compris respectivement pour `POSIX` et `MPI I/O` entre :

$$11 < \frac{T_{POSIX\,sur\,NFS}}{T_{POSIX\,sur\,NFS\,aIOLi}} < 50 \qquad \text{et} \qquad 3,5 < \frac{T_{MPII/O\,sur\,NFS}}{T_{POSIX\,sur\,NFS-aIOLi}} 6,5$$

Les performances atteintes pour des comportements parallèles se révèlent globalement meilleures que celles fournies par la lecture séquentielle du fichier (courbe «`POSIX aIOLi - 32`» *vs* courbe référence «`POSIX - 1`»). Cela s'explique par le fait que le service *aIOLi* «accède» au fichier à une granularité 32 fois supérieure à celle de l'application séquentielle. Lorsque l'application «`POSIX-1`» accède à un bloc de 8Ko, l'application parallèle «`POSIX aIOLi - 32`» génère 32 requêtes. Ces requêtes sont ensuite agrégées par *aIOLi* en une requête virtuelle de 256Ko ($8Ko * 32$).

Les seuls cas où la lecture séquentielle est plus efficace, sont pour les granularités de 64 Ko et de 512 Ko. Ces granularités correspondent au seuil pour lesquels l'algorithme proposé dans

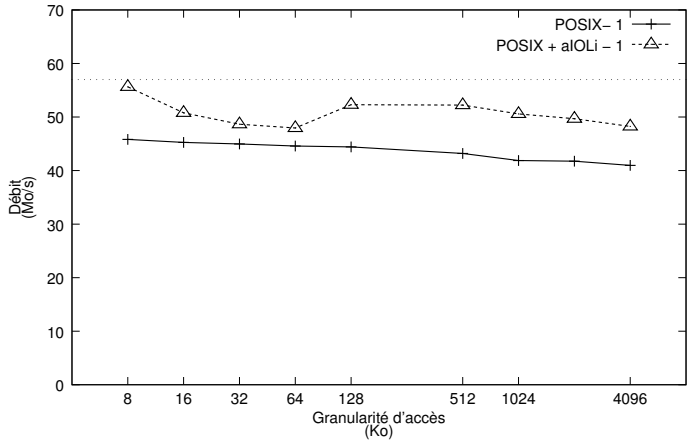


FIG. 7.6 – Comportement séquentiel - Lecture - `NFS` *vs* `NFS`-*aIOLi*
Comparaison des performances lors de la récupération d'un fichier de 4 Go par un seul processus.

aIOLi ne peut plus détecter la totalité du schéma d'accès parallèle et exploite uniquement la politique d'ordonnancement à base de fenêtres dédiées. En effet, dès que la granularité d'accès est plus grande que la granularité de `NFS` (fixée à 32 Ko), l'ensemble des requêtes transmises depuis les processus de l'application parallélisée ne forme plus une seule requête virtuellement contiguë. Le comportement est alors comparable au type 3 du jeu de tests b_eff_io (cf. figure 7.2). Dans un tel cas, plus la granularité devient importante, plus le système à base de fenêtres est efficace.

La seconde expérience a consisté à évaluer les apports de la stratégie d'ordonnancement sur un comportement séquentiel. La figure 7.6 présente les résultats.

Le service *aIOLi* permet d'atteindre là encore de meilleures performances. Les algorithmes de pré-chargement côté serveur sont très sensibles et il est impératif que les accès arrivent de manière contiguë [BGH05]. Or, lorsque la stratégie *read-ahead* est activée sur le nœud client, plusieurs accès sont transmis par le démon `rpciod` et réceptionnés par les *threads* `nfsd`. L'ordre de traitement des requêtes entre les différents démons n'est pas strict et les politiques de pré-chargement sont sans cesse activées puis désactivées. Le service *aIOLi*, qui sérialise les accès, garantit sur la totalité de l'opération une meilleure contiguïté et donc de meilleures performances (les stratégies de pré-chargement restant actives sur de plus longues durées).

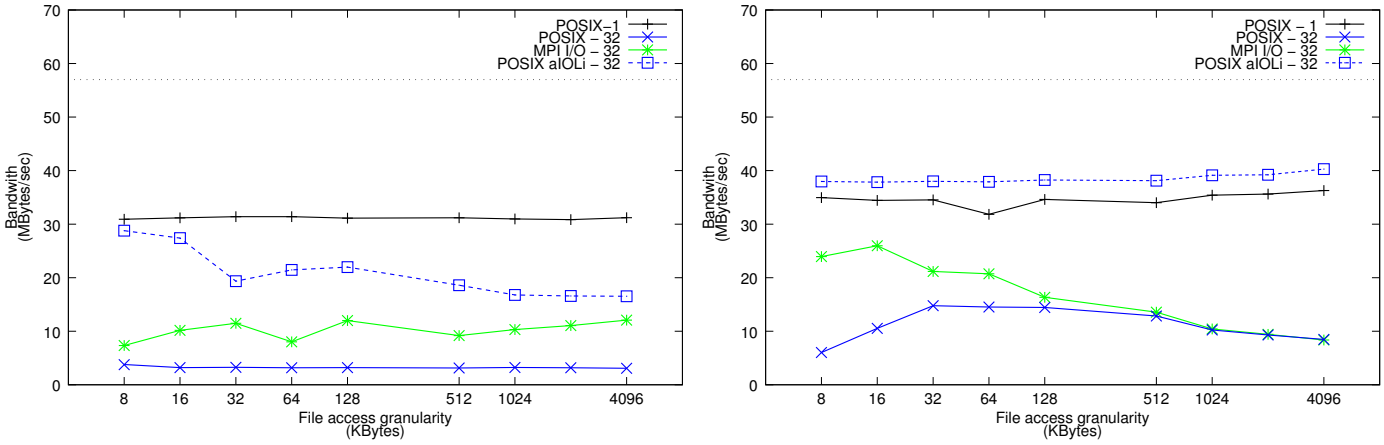
D'un point de vue global et pour les accès en lecture, les mécanismes de détection des comportements parallèles et la bonne utilisation des techniques de pré-chargement permet à notre système d'atteindre des performances relativement proches du débit maximum fourni par le disque.

Écriture

Dans cette partie, nous présentons les résultats obtenus lors d'une décomposition en écriture. Les paramètres restent identiques : 32 clients manipulent un fichier de 4 Go via le jeu de tests *IOR*.

Comme nous l'avons mentionné, les performances en écriture sont très différentes selon les paramètres de montage des systèmes de fichiers. De plus, pour des fichiers de tailles conséquentes, il semble primordial de pré-allouer l'espace.

Les courbes que nous allons décrire permettent de comparer les interfaces `POSIX` et `MPI I/O` dans les cas les plus favorables pour les performances : le cas «4» pour un mode d'écriture sans pré-allocation (client `async` et serveur `async`) et le cas «2» (client `async` et serveur `sync`) lorsque le fichier est préalablement alloué (cf. section 4.2). Les expériences ont été conduites sur les deux architectures : `NFS` et `NFS`-*aIOLi*. La figure 7.7 illustre les résultats.



(a) - Sans pré-allocation

(b) - Avec pré-allocation

FIG. 7.7 – Décomposition d'un fichier de 4 Go sur 32 processus - Écriture - `NFS` *vs* `NFS`-*aIOLi*
Comparaison des performances obtenues par l'utilisation de l'interface `POSIX`, `MPI IO` et `POSIX` au dessus d'un serveur `NFS`-*aIOLi*. Le fichier est écrit en pré-allouant ou non l'espace de stockage.

Même si les apports sont moins significatifs que pour les lectures, l'utilisation du service *aIOLi* est positive dans les deux cas. Les facteurs d'amélioration sont respectivement pour `POSIX` et `MPI I/O` :

– lorsque le fichier est écrit sans pré-allouer l'espace

$$5,2 < \frac{T_{POSIX\,sur\,NFS}}{T_{POSIX\,sur\,NFS\,aIOLi}} < 8,5 \qquad \text{et} \qquad 1,3 < \frac{T_{MPII/O\,sur\,NFS}}{T_{POSIX\,sur\,NFS-aIOLi}} < 3,9$$

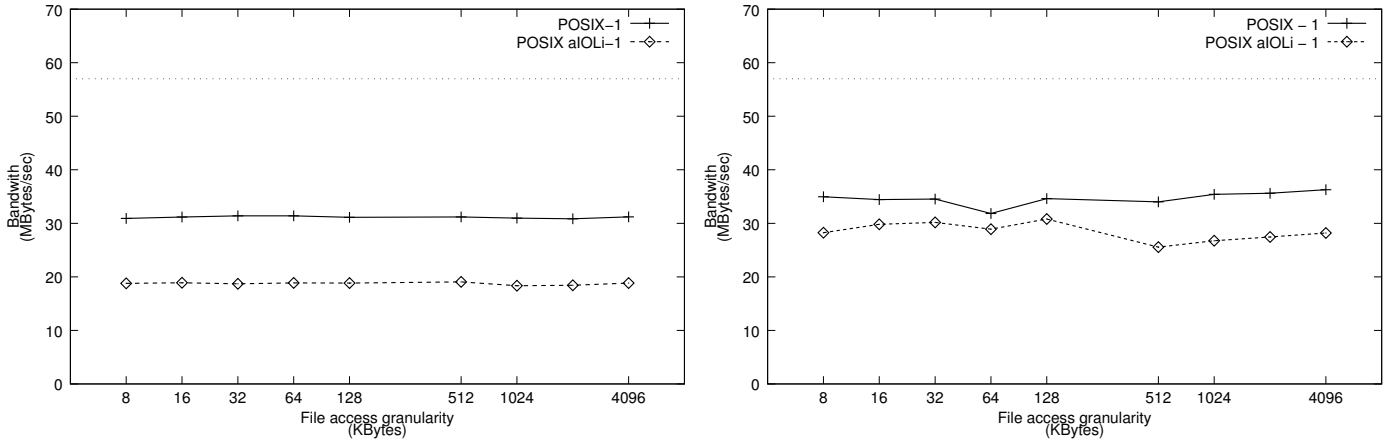
– pour le cas où l'espace est pré-alloué,

$$2,5 < \frac{T_{POSIX\,sur\,NFS}}{T_{POSIX\,sur\,NFS\,aIOLi}} < 6,5 \qquad \text{et} \qquad 1,5 < \frac{T_{MPII/O\,sur\,NFS}}{T_{POSIX\,sur\,NFS-aIOLi}} < 5$$

Dans ce dernier cas, les performances atteintes en parallèle grâce au service *aIOLi* sont similaires à celles obtenues par le comportement séquentiel. Cela s'explique par le fait que les optimisations effectuées par *aIOLi* permettent de régénérer en grande partie l'accès séquentiel. Cependant, à la différence des accès en lecture qui profitent d'une granularité d'accès «cumulée» la taille d'accès en écriture ne semble pas influer sur les performances dans le cas d'un comportement séquentiel et les améliorations sont moins significatives.

De plus, la sérialisation des écritures ajoute des délais et il est nécessaire d'avoir un nombre de requêtes suffisamment important pour que le gain devienne visible. La figure 7.8 illustre

tout à fait ces propos. Nous comparons les comportements synchrones au dessus des deux architectures et pour les deux modes.



(a) - Sans pré-allocation

(b) - Avec pré-allocation

FIG. 7.8 – Comportement séquentiel - Écriture - `NFS` *vs* `NFS`-*aIOLi*
Comparaison des performances lors de la récupération d'un fichier de 4 Go par un seul processus. Le fichier est écrit en pré-allouant ou non l'espace de stockage.

Dans le cas où la pré-allocation n'a pas été effectuée et sur un serveur exportant une partition `ext3` en asynchrone (graphique (a)), un surcoût d'environ 40% apparaît. La stratégie d'écriture retardée mise en place pour la partition `ext3` semble en être la principale cause. Dans le second cas où la partition est montée en synchrone (graphique (b)), le surcoût est moindre, il varie entre 10% et 20%. Toutefois, ces résultats doivent être approfondis afin de comprendre l'impact de la phase de pré-allocation dans le surcoût : les résultats fournis par une expérimentation conduite sur un serveur «synchrone» sans pré-allocation n'ayant pas révélé de surcoût.

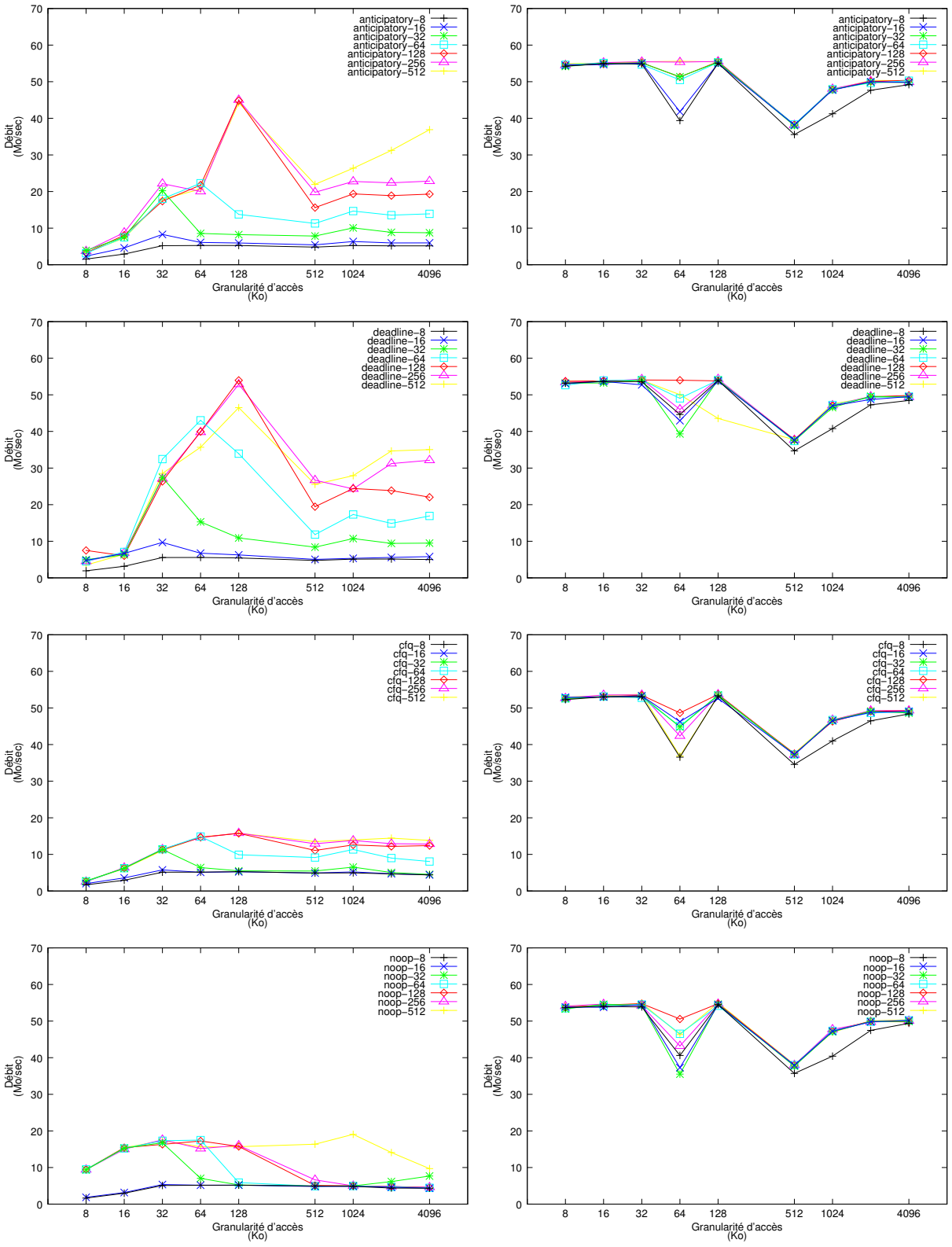
7.2.2. *aIOLi*, impact sur les ordonnanceurs bas niveau

Cette dernière expérimentation en contexte mono-applicatif a été réalisée pour deux principaux buts : d'une part, l'étude de l'impact de la sérialisation sur les ordonnanceurs d'E/S bas niveau, et, d'autre part, l'évaluation de l'importance du nombre de *threads* associé au service *aIOLi* sur les performances.

La figure 7.9 représente les valeurs mesurées lors d'une décomposition en lecture sur 32 processus déployés sur 32 nœuds. La colonne de gauche reprend les valeurs décrites en section 4.3.2.

La première observation concerne les performances qui sont relativement similaires pour l'ensemble des stratégies d'ordonnancements dans le cadre du système `NFS`-*aIOLi*. La sérialisation semble imposer la stratégie mise en œuvre au sein de notre service. Les requêtes étant transmises une à une, les ordonnanceurs bas niveau n'ont pas la possibilité d'influer sur les décisions retenues à plus haut niveau.

Par ailleurs, mis à part pour les granularités de 64 ko et 512 Ko, les performances maximales sont directement atteintes à partir d'un nombre réduit de *threads* `nfsd`.



NFS NFS-*aIOLi*

FIG. 7.9 – Impact de la sérialisation et apport du nombre de `threads`

Evaluation des performances pour une décomposition en lecture de 4 Go par 32 processus répartis sur 32 nœuds. Les architectures `NFS` (à gauche) et `NFS`-*aIOLi* sont comparées.

7.2.3. Bilan

Cette partie a permis de vérifier que les mécanismes d'agrégation présents dans la stratégie d'ordonnancement global permettent de répondre aux attentes des applications parallélisées fortement dépendante des E/S. Les apports pour les opérations de type lecture sont significatifs. Les optimisations mises en œuvre permettent pour la plupart des granularités d'atteindre un débit proche du maximum que puisse délivrer l'unité de stockage. Par exemple, la sérialisation permet d'affiner les comportements séquentiels, il en résulte une meilleure exploitation des mécanismes de pré-chargement et donc des meilleures performances.

Concernant les écritures, les objectifs pour les comportements parallèles sont en grande partie validés en fournissant des performances supérieures aux interfaces `POSIX` et `MPI I/O`. Cependant, comme nous l'attendions, notre proposition ajoute un délai qui devient très vite significatif lorsque le serveur exploite des stratégies d'écriture retardée. Il est alors nécessaire d'avoir un nombre important de requêtes dans le service afin de recouvrir ces délais et fournir un gain visible. Les comportements synchrones sont ceux qui souffrent le plus de ce phénomène. Ces résultats doivent être approfondis enfin de déterminer où apparaît la perte et proposer une éventuelle solution.

Dans la partie suivante, nous allons nous intéresser au comportement du service *aIOLi* (et de la stratégie d'ordonnancement proposée) dans un contexte multi-applicatif.

7.3. Évaluation multi-applicative

Après avoir évalué le surcoût de la sérialisation et validé les mécanismes mis en place dans la variante *MLF* pour l'optimisation des comportements d'E/S parallèles au sein d'une application, nous allons aborder un de nos principaux objectifs : la prise en compte des aspects multi-applicatifs.

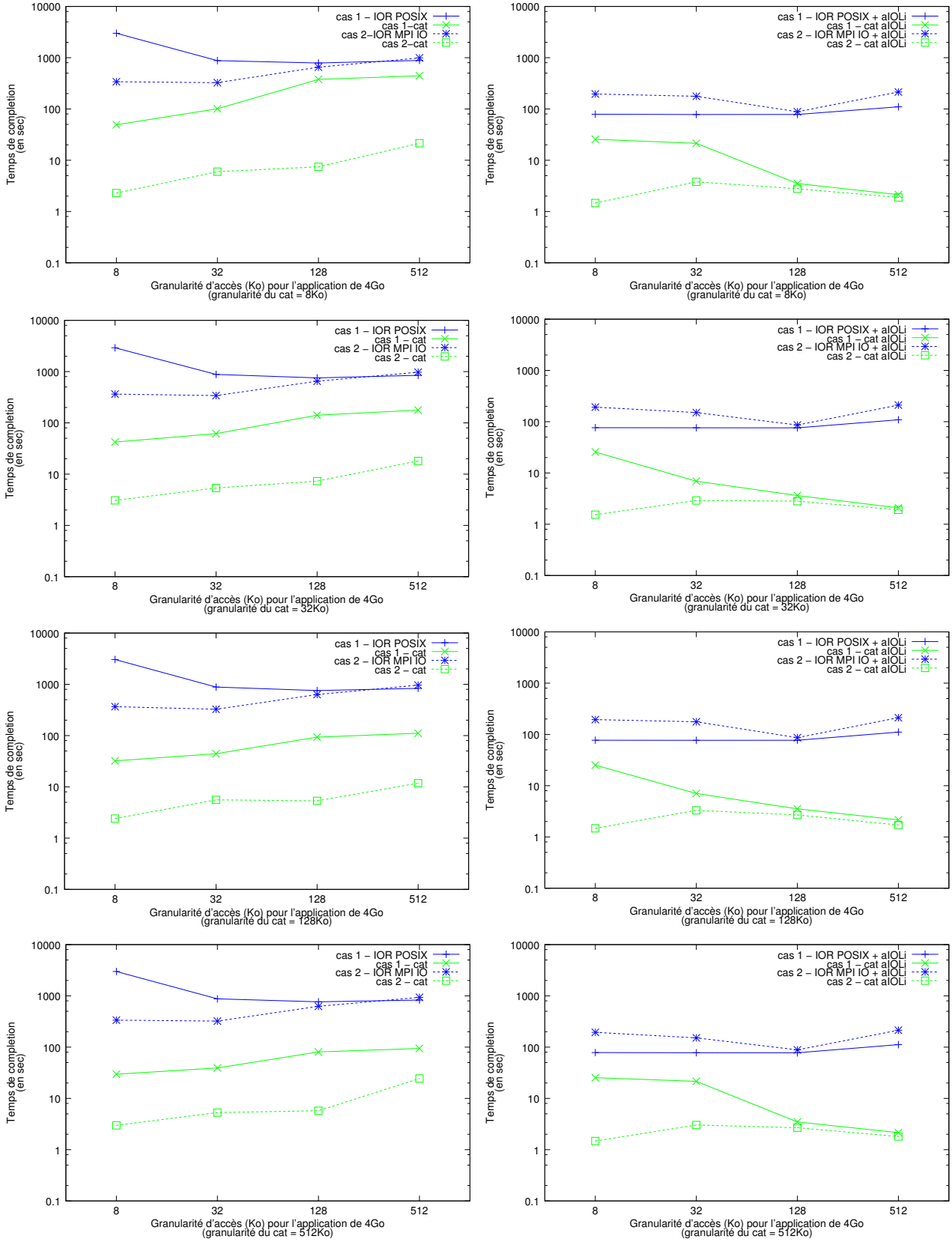
Les expériences décrites dans la section 4.4 ont été reconduites sur un serveur `NFS` combiné au service *aIOLi*. Les deux premières évaluations traitent de l'impact d'un mode d'accès parallèle sur une autre application. La troisième, quant à elle, permet d'analyser les performances délivrées par un serveur `NFS`-*aIOLi* en présence de 10 applications concurrentes.

Pour les deux premières expériences, seules les performances lors d'accès de type lecture sont décrites et analysées. Plusieurs tests similaires ont été réalisées en écriture afin d'étudier l'impact. Cependant, une étude plus approfondie est nécessaire pour comprendre l'implication de chacun des paramètres et fournir une analyse rigoureuse des résultats.

7.3.1. Impact d'une décomposition de 4 Go sur une opération «cat-like» brève

Le but de ce paragraphe est de déterminer si un service de contrôle, de réordonnnancement et de régulation global comme celui que fournit *aIOLi* permet de diminuer l'impact engendré par une application fortement dépendante des E/S sur une application faiblement dépendante du système de stockage.

Les conditions d'expérimentation restent identiques à celles décrites en section 4.4.1 : le jeu de tests *IOR* est utilisé pour réaliser la décomposition parallèle de 4 Go par 32 processus répartis sur 32 nœuds et un programme de type `cat` de 16 Mo est exécuté sur un $33^{\text{ème}}$ nœud 15 secondes après le départ de la décomposition. Les nouveaux résultats obtenus sur l'architecture `NFS`-*aIOLi* sont comparés à ceux précédemment mesurés sur une architecture à base de serveur `NFS` traditionnel. Les graphiques apparaissent à la figure 7.10.



NFS NFS-*aIOLi*

FIG. 7.10 – Impact d'une décomposition de 4Go sur une opération «`cat`-*like*» de 16Mo
Comparaison entre les performances fournies par un serveur NFS et un serveur NFS exploitant le service *aIOLi*

Les graphiques de gauche présentent les performances atteignables sur un serveur NFS traditionnel tandis que ceux de droite donnent les valeurs mesurées sur un serveur NFS-*aIOLi*. Enfin, les courbes en pointillées correspondent à l'utilisation des routines MPI I/O pour l'application parallèle (nous rappelons que le programme séquentiel n'utilise que les routines POSIX : open, read, close). Les expériences ont été exécutées pour une granularité d'accès pour le cat de 8 Ko, 32 Ko, 128 Ko et 512 Ko.

Comme nous pouvons le constater, la granularité d'accès du programme de type cat ne semble pas avoir de réelle importance et quelque soit la granularité utilisée pour l'application parallèle, l'usage du service *aIOLi* est bénéfique. Lorsque la stratégie ne peut plus détecter le schéma d'accès parallèle et générer une seule et même requête virtuelle [1] , l'application de type cat se comporte exactement comme un des processus MPI du jeu de tests *IOR* et peut à son tour profiter du mécanisme de fenêtres dédiées : son temps de complétion atteint une valeur acceptable d'environ 2 secondes (nous rappelons, qu'en théorie, moins d'une seconde devrait être nécessaire pour récupérer un fichier de 16 Mo pour un disque pouvant générer 57 Mo/s).

D'un point de vue global à l'interface POSIX, pour les petites granularités de l'application *IOR*, le service *aIOLi* améliore de près de 2 fois les performances de l'application cat-*like* et de plus de 40 fois celles du programme fortement dépendant des E/S. Les performances atteintes sont proches du maximum avec un temps de complétion autour de 78 secondes. Le meilleur temps pouvant être proposé par l'unité de stockage lors d'un accès local est, nous le rappelons, de 72 secondes pour la manipulation d'un fichier de 4 Go. Pour des granularités plus importante, le service *aIOLi* optimise les performances pour les deux applications d'environ 8 fois pour le programme parallèle et d'au moins 70 fois pour celui séquentiel (un facteur d'amélioration supérieur à 200 est atteint lorsque le cat est réalisé à une granularité de 8 Ko).

Enfin, même si l'utilisation des routines MPI I/O permet d'atteindre de manière immédiate de bonnes valeurs pour l'application de type cat sur l'architecture NFS-*aIOLi*, elle ajoute un surcoût significatif pour l'application parallèle (courbes de droite «cas 1 - IOR POSIX + aIOLi» *vs* «cas 2 - IOR MPI IO + aIOLi»).

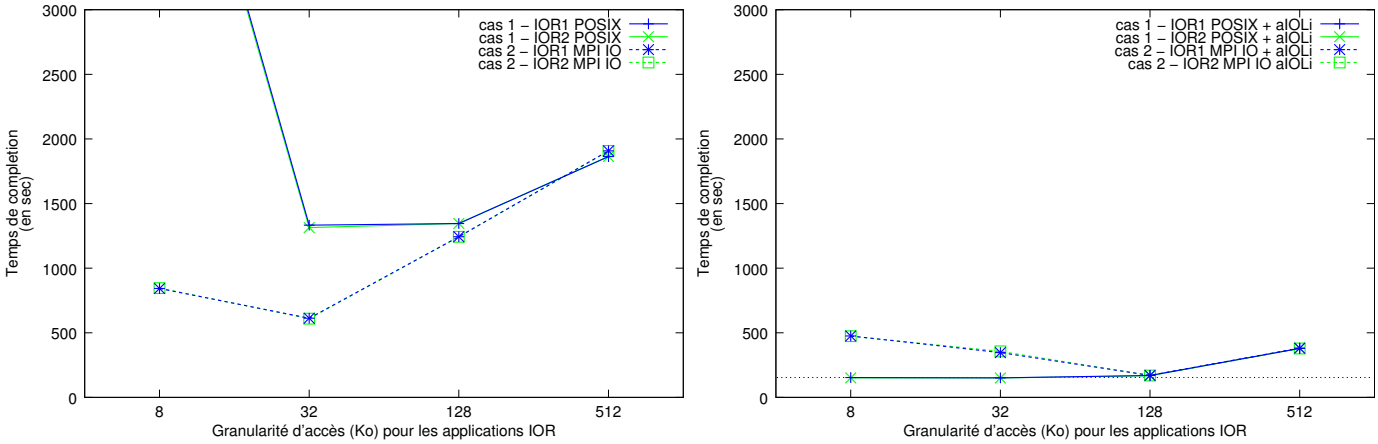
L'utilisation du service *aIOLi* se révèle plus qu'avantageuse dans le cas que nous venons d'aborder. Avant de traiter une expérience représentant une charge éventuelle d'une grappe (10 applications), nous allons analyser le comportement de la variante *MLF* lors de deux décompositions parallèles.

7.3.2. Impact mutuel de deux applications bornées par les E/S

Nous avons montré que la stratégie incluse dans le service *aIOLi* permettait de réduire l'impact engendré par une décomposition parallèle sur une application ne dépendant pas réellement du système de stockage. Les performances de chacune des applications sont améliorées de manière significative. Dans ce paragraphe, nous allons nous attarder sur l'impact mutuel entre deux décompositions de 4 Go. Deux instances du programme *IOR* (2*32 processus MPI déployés sur 2*32 nœuds) ont été lancées de nouveau sur une architecture basée sur un serveur NFS-*aIOLi*. Les performances obtenues ont été comparées avec celles décrites en section 4.4.2.

La figure 7.11 illustre cette comparaison : le temps de complétion pour chacune des deux applications en fonction de la granularité d'accès est présenté. Dans ce cas, nous nous intéressons uniquement à la confrontation des valeurs mesurées lors de l'utilisation de l'interface POSIX puis MPI I/O sous les deux architectures. Nous avons choisi de supprimer l'échelle logarithmique sur l'axe des ordonnées.

[1] La granularité d'accès devient supérieure à la granularité du protocole du système de fichiers, cf. section 7.2.1.



NFS NFS-*aIOLi*

FIG. 7.11 – Impact mutuel entre deux décompositions de 4 Go
Comparaison entre les performances fournies par un serveur NFS (à gauche) et un serveur NFS-*aIOLi* (à droite)

Une fois encore, la stratégie d'ordonnancement permet d'améliorer considérablement les performances. Comme nous l'avions fait remarquer, même si la stratégie collective de MPI I/O permet de réduire le temps de complétion de chacune des applications pour des petites granularités, elle souffre rapidement de la concurrence entre les accès et ne permet plus d'améliorer les performances pour les grosses granularités (courbes de gauche).

Lors de l'utilisation du service *aIOLi* et pour les petites granularités, les performances sont proches de l'optimal : chacune des applications termine la récupération de son fichier de 4 Go dans un délai de 150 secondes. Or pour pouvoir lire 8 Go sur l'unité de stockage, il faut un peu moins de 144 secondes à un débit de 57 Mo/s.

L'écart de temps est considérable entre les différentes solutions : l'approche POSIX au dessus d'un serveur traditionnel requiert plus de 1 heure et 30 minutes, l'utilisation de l'approche collective proposée par la bibliothèque *ROMIO* permet de réduire ce temps à 14 minutes. L'usage de notre service permet d'atteindre 2 minutes et 33 secondes et ce sans exploiter une interface spécifique ou même recompiler le code des applications.

Pour une granularité supérieure à 32 Ko, la stratégie n'exploite plus que les optimisations à base de fenêtre ce qui explique la faible dégradation. Néanmoins, les performances restent bien meilleures que l'approche POSIX ou bien MPI I/O.

Par ailleurs, chacune des applications termine dans des temps comparables, le plus gros écart de complétion étant de 2 secondes (ce qui explique que les courbes se recouvrent). L'équité est donc bien assurée par la stratégie et aucune des deux applications n'est favorisée. Il serait tout à fait possible de définir un accroissement de la taille de la fenêtre spécifique à chacune des applications afin d'en privilégier une au dépend de l'autre. Pour le moment, la manière dont est fixée la durée de chaque fenêtre est identique pour chacune des applications. Cela dépend du nombre de requêtes en attente de traitement et du pourcentage d'utilisation de la dernière fenêtre attribuée à ce fichier.

Enfin, l'utilisation des routines MPI I/O ne semble pas être intéressante lorsque le serveur NFS exploite le service *aIOLi*. Un important surcoût lié au mécanisme de synchronisation employé dans l'approche collective apparaît pour les petites granularités.

L'algorithme basé sur l'approche *MLF* associé à l'attribution dynamique de la taille de chaque fenêtre de régulation augmente de manière significative les performances globales du

système de stockage dans le cadre de deux applications fortement dépendantes des E/S et accédant selon des modes parallèles aux fichiers. Dans le paragraphe suivant, nous allons évaluer l'apport de cette stratégie lorsque dix applications indépendantes et différentes sont exécutées en concurrence sur la grappe. Cette expérience va nous permettre de montrer d'une part l'intérêt d'un service d'ordonnancement «de plus haut niveau» et d'autre part certains aspects devant être approfondis afin de valider totalement ce type de solution.

7.3.3. Haut niveau de concurrence : 10 applications, contribution d'*aIOLi*

Un des principaux objectifs des travaux conduits tout au long de cette thèse a été de proposer une solution transparente permettant un contrôle, un réordonnancement et une régulation de l'ensemble des E/S au sein d'une grappe.

Dans la section 4.4.3, nous avons analysé un jeu de tests composé de 10 applications afin d'évaluer les performances d'un système `NFS` en présence d'un large éventail de comportements pouvant survenir sur une grappe pendant une période. Cette évaluation nous avait permis de montrer l'impact considérable sur les performances et l'inefficacité des stratégies `MPI I/O` dans un tel contexte. Certes l'utilisation des fonctions collectives permettait aux applications séquentielles de manipuler leurs données dans des temps moins importants que ceux requis lors de l'usage des routines `POSIX` mais cela au dépend d'une dégradation significative pour les applications exploitant des schémas d'accès parallèles.

Le tableau 7.2 qui reprend ces valeurs a été complété avec les mesures effectuées pour le même jeu de tests sur un serveur `NFS`-*aIOLi*. La figure 7.12 fournit également une présentation des valeurs sous forme d'histogrammes.

Identifiant de l'application	Description	Temps de complétion			
		`NFS`		`NFS`+*aIOLi*	
		`POSIX`	`MPI I/O`	`POSIX`	`MPI I/O`
1	`read` decompos. - 2Go (32 nœuds, granularité=128Ko)	490	840	134	500
2	`write` decompos. - 2Go (32 nœuds, granularité=128ko)	409	815	107	604
3	`read` decompos. - 256Mo (16 nœuds, granularité=8ko)	595.5	728	104	415
4	`write` decompos. - 128Mo (8 nœuds, granularité=64ko)	51	257	14.5	247
5	`read` sequentiel - 32Mo (1 nœud , granularité=4ko)	531	9	48.5	3
6	`write` sequentiel - 32Mo (1 nœud, granularité=4ko)	208	9	47	6
7	`read` sequentiel - 4Mo (1 nœud, granularité=32ko)	57	1.5	6	1
8	`write` sequentiel - 4Mo (1 nœud, granularité=32ko)	39	2	19	2
9	`read` sequentiel - 1Go (1 nœud, granularité=2Mo)	558	59	143.5	54
10	`write` sequentiel - 512Mo (1 nœud, granularité=2Mo)	192	71	84	61.5

Les temps sont exprimés en secondes.

TAB. 7.2 – Temps de complétion de 10 applications indépendantes s'exécutant en concurrence
Comparaison entre les performances d'un serveur NFS (abordées en section cf. 4.4.3) et d'un serveur `NFS`-*aIOLi*

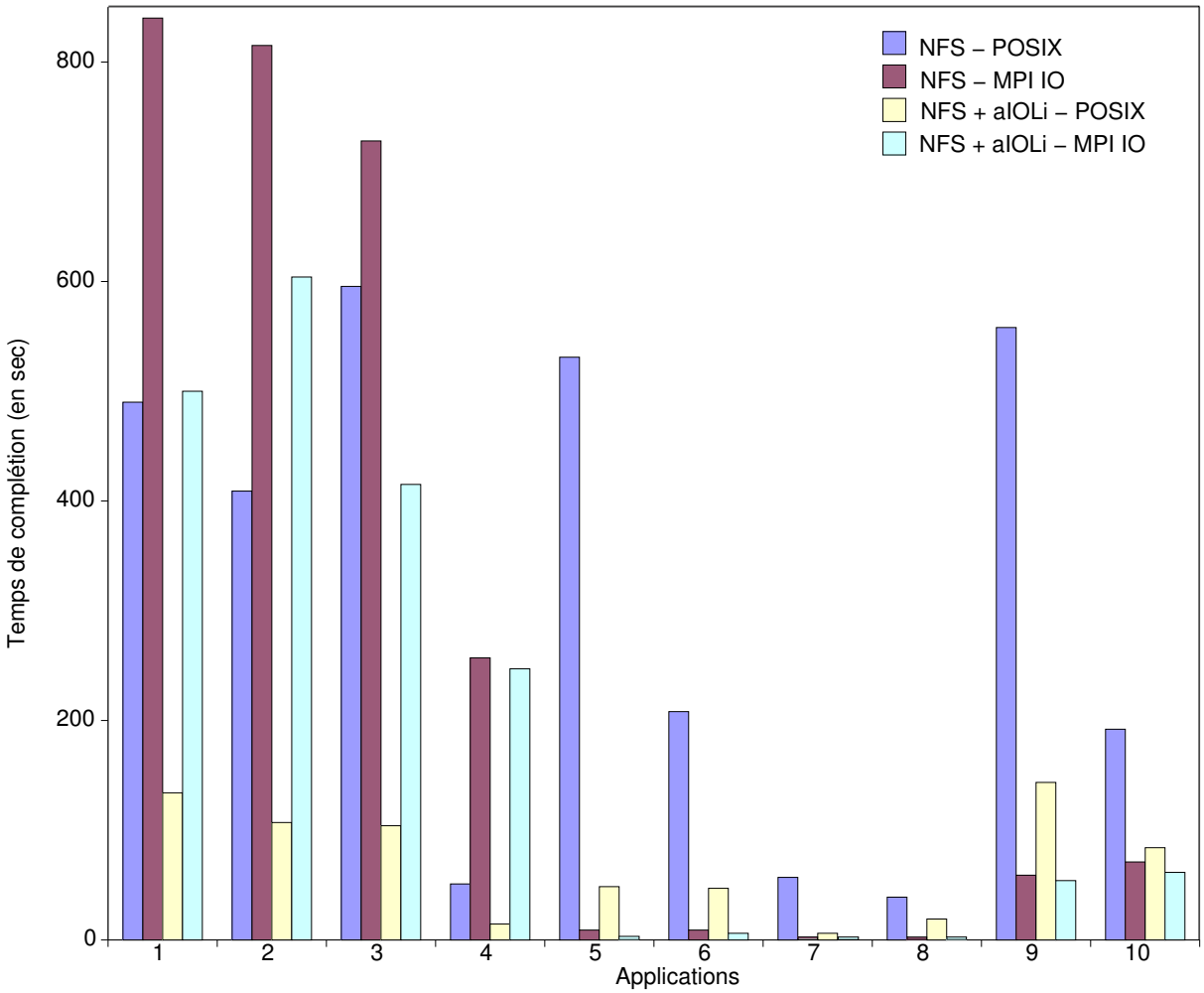


FIG. 7.12 – Temps de complétion de 10 applications indépendantes s'exécutant en concurrence
Pour chacune des applications les performances pour les 4 cas sont présentées : cas 1 et cas 2 serveur traditionnel `NFS` avec les interfaces `POSIX` puis `MPI I/O`; cas 3 et 4 *idem* mais sur un serveur `NFS` exploitant le service *aIOLi*.

L'analyse de l'ensemble de ces valeurs est réalisée de manière analogue à celle effectuée en section 4.4.3. Dans un premier temps, nous allons considérer le jeu de tests comme une unique et même application manipulant des données en lecture et écriture sur 6 Go. Dans un tel cas, l'utilisation du service *aIOLi* combiné aux routines `POSIX` permet d'améliorer les performances d'un facteur supérieur à 4 en comparaison avec les mesures `POSIX` et de presque 6 par rapport à `MPI I/O` pour un serveur `NFS` traditionnel. Dans ce cas précis, la solution *aIOLi* permet d'être à 33% de l'optimal des performances pouvant être fournies par l'unité de stockage (108 secondes étant nécessaires au disque pour manipuler 6 Go).

L'utilisation des routines `MPI I/O` au-dessus du serveur `NFS`-*aIOLi* ne semble pas être un choix judicieux du point de vue de «la borne supérieure» comparable à l'utilisation de l'interface `POSIX` sur un serveur traditionnel (604 sec *vs* 595.5 sec).

En analysant de manière plus fine les temps de complétion de chacune des applications dans le cas `NFS`-*aIOLi* `POSIX`, nous pouvons remarquer que sauf pour l'application 4, les temps de complétion semblent être liés à la taille des données manipulées ainsi qu'à la granularité d'accès : la stratégie *aIOLi* semble donc proposer un bon compromis performance *vs* équité. Nous pouvons également noter que le facteur d'amélioration est d'autant plus important lorsque

la granularité d'accès est inférieure à celle du protocole du système de fichiers (32 ko) pour les accès parallèles (les techniques d'agrégation pouvant être mises en œuvre). Lorsque les granularités sont supérieures, seul le système de fenêtre permet d'améliorer les performances.

Pour ce qui est du temps de complétion de l'application 4, il est nécessaire d'approfondir l'étude des écritures au sein du protocole NFS. En effet, le temps de complétion proposé par la solution standard POSIX NFS était déjà considérablement inférieur à celui des autres écritures (à proportion équivalente). Comme nous l'avions mentionné, chacun des fichiers en écriture est pré-alloué et les paramètres de montage sont en asynchrone côté serveur et côté client. L'impact des stratégies de type écriture retardée doit être approfondi pour le modèle standard dans un premier temps et avec la sérialisation par la suite.

Dans tous les cas et d'un point de vue général, l'approche POSIX sur le serveur NFS-*aIOLi* améliore les performances de chacunes des applications par rapport aux valeurs fournies par la même interface sur un serveur NFS que ce soit en écriture ou en lecture, le plus petit facteur étant de 2 pour l'application 8 et le plus grand proche de 11 pour l'application 5.

En ce qui concerne l'interface MPI I/O au dessus du serveur NFS traditionnel, les fonctions standard POSIX au-dessus du serveur NFS-*aIOLi* sont plus performantes que les routines collectives exploitées par *ROMIO*. Les délais d'inactivité induits par les mécanismes de synchronisation permettent aux applications séquentielles d'avoir des temps de complétion nettement inférieurs à ceux proposés par l'approche POSIX NFS-*aIOLi*.

Cependant la granularité d'accès pour les applications et la borne d'interactivité que nous avons définies [2] , limitent le gain pour les applications séquentielles. Ainsi, lorsque la granularité est faible, par exemple pour l'application 5, elle est de 4 Ko, peu de requêtes sont transmises au serveur durant la première «fenêtre» initialisée à QB (cf. section 5.3.1) : la granularité étant de 4 Ko, une requête de 32 Ko du serveur NFS permet de servir côté client 8 accès. De ce fait, la fenêtre n'est pas pleinement exploitée et le service *aIOLi* ne propose pas un délai plus important. En jouant sur le paramètre QB et la taille de la fenêtre maximale, les applications séquentielles faiblement dépendantes seront favorisées au dépend des applications parallèles fortement dépendantes.

7.4. Bilan

Dans ce chapitre, nous avons évalué :

- le surcoût et l'impact de la centralisation des accès au sein du service *aIOLi* ;
- les mécanismes de détection et d'agrégation des schémas d'accès parallèles ;
- le comportement et les apports de la stratégie d'ordonnancement global dans un contexte multi-applicatif.

En ce qui concerne la sérialisation des requêtes au sein du service *aIOLi* et le surcoût qu'elle engendre, le test Bonnie ++ généralement utilisé pour analyser les performances d'un système de fichiers local sous plusieurs modes d'accès a été exécuté sur un serveur NFS standard puis sous un serveur NFS exploitant le service *aIOLi*. Les performances obtenues pour l'exécution d'un client puis de quatre se sont révélées pour la plupart bonnes.

Cependant et comme nous l'attendions, un surcoût pour les écritures séquentielles d'environ 10% a été mesuré. Ce surcoût spécifique aux écritures séquentielles est dû au passage

[2] La taille de la fenêtre de régulation a été fixée à 500 ms.

dans le service *aIOLi* qui n'apporte aucune optimisation. Ce phénomène est considérablement amplifié par une stratégie de type écriture retardée lorsque celle-ci est mise en œuvre côté serveur. L'évaluation d'une écriture séquentielle d'un fichier de 4 Go dans la section 7.2.1.2 nous a permis d'identifier clairement ce problème.

Pour le surcoût dû au passage dans le service *aIOLi*, il ne semble pas possible de l'éliminer. Cependant, plus le nombre d'applications s'exécutant sur la grappe est important plus la stratégie à base de fenêtre dédiée permet d'atteindre de meilleures performances pour chacune d'entre-elles (le surcoût disparaît au profit d'améliorations significatives). La dernière expérience constituée de 10 applications nous a montré un tel comportement : chacune des applications réalisant des écritures séquentielles a été améliorée d'au moins un facteur 2.

Néanmoins, il semble intéressant d'approfondir le problème d'une unique application en écriture séquentielle afin de déterminer si un mécanisme quelconque pourrait permettre de pallier les contraintes induites par une politique de type écriture retardée. La principale contrainte est que notre service de contrôle, de réordonnancement et de régulation n'interagit avec le système de fichiers que par les appels au fonctions `process_request` et `process_requests`. Ainsi, aucune fonction n'a été prévue afin de déterminer la stratégie de cache mise en place pour les écritures. Toutefois, même si de telles fonctionnalités étaient proposées, nous insistons sur le fait qu'elles seraient utiles pour le cas unique où une seule application réalisant des écritures séquentielles est en cours d'exécution dans toute la grappe. Il serait intéressant d'étudier si un tel cas intervient régulièrement ou non sur une grappe.

Pour les lectures séquentielles, la sérialisation permet d'affiner l'ordre de traitement des requêtes `NFS` et il en résulte une utilisation plus efficace des techniques de pré-chargement et donc de meilleures performances (cf. section 7.2.1.1).

Afin d'examiner l'impact de la sérialisation sur le facteur du passage à l'échelle, nous avons exécuté le jeu de tests `b_eff_io`. Le nombre de processus utilisé a varié de 2 à 96. Pour l'ensemble des schémas d'accès parallèles évalués, l'utilisation du service *aIOLi* s'est révélée bénéfique. Nous avons d'ailleurs observé qu'un nombre élevé de processus dans le jeu de tests est favorable au service *aIOLi*. De plus, le programme `b_eff_io` étant développé au-dessus de la bibliothèque *ROMIO*, les améliorations pouvant être apportées par la stratégie d'ordonnancement sont limitées et un surcoût lié aux mécanismes de synchronisation est rajouté dans plusieurs cas. Par exemple, les performances en lecture atteignent difficilement un débit de plus de 15 Mo/s. Or, les tests réalisés avec l'application *IOR* [3] sous l'interface `POSIX` ont révélé qu'il était possible d'atteindre un débit de plus de 50 Mo/sec pour un accès de type 3.

En ce qui concerne la détection et l'agrégation des schémas d'accès parallèles, les expériences conduites dans un contexte mono-applicatif ont permis de valider les mécanismes intégrés dans la stratégie d'ordonnancement proposée que ce soit pour les accès de type lecture ou bien écriture. Toutefois, il semble qu'une opération de pré-allocation soit requise afin d'obtenir des améliorations plus significatives pour les écritures.

D'un point de vue général à l'ensemble des remarques qui a été réalisé sur les écritures et même si les débits atteints sont plus élevés, une étude plus spécifique de chacun des paramètres (synchrone/asynchrone, pré-allocation) permettra de mieux comprendre le rôle de chacun dans les performances en écriture.

[3] En comparaison avec le programme `b_eff_io`, le jeu de tests *IOR* évalue les types 0, 1, 3 et 4 selon l'utilisation des routines sélectionnées (cf. section 7.1.2).

Pour terminer, l'évaluation autour d'un jeu de tests composé de dix applications a montré sans équivoque l'intérêt d'un service de contrôle, d'ordonnancement et de régulation des requêtes au sein d'une architecture parallèle basée sur un serveur `NFS` pour partager les données.

Nous avons vu l'importance des paramètres QB et de la taille maximale de la fenêtre. Ces deux valeurs définissent les contraintes d'interactivté et d'équité souhaitées dans notre stratégie. En jouant sur ces valeurs, il est possible de proposer une qualité de service plus fine en favorisant un type d'application ou un autre.

Le chapitre suivant présente un bilan global de l'ensemble des travaux réalisés. A la différence de la conclusion qui sera plus générale et qui ouvrira les travaux vers différentes perspectives, la synthèse qui va être réalisée lors du prochain chapitre va se concentrer de manière spécifique sur le service *aIOLi*. Un bilan technique puis un inventaire des travaux en cours ou à finaliser vont être effectués.

BILAN GÉNÉRAL 8

Dans la première partie de ce document, nous avons abordé l'écart de performances entre les puissances d'analyse et les débits fournis par les unités de stockage présentes dans les architectures parallèles de type grappe. Les modes d'accès généralement utilisés par les applications *HPC* ont été également évoqués. Ces schémas d'accès parallèles dégradent de manière significative les performances des E/S, amplifiant ainsi l'écart de performances entre les ressources calcul et celles liées au stockage. Plusieurs solutions ont été proposées afin d'améliorer les performances au sein d'une application. Cependant, nous avons vu en début de deuxième partie que les approches disponibles ne sont pas appropriées à un contexte multi-applicatif comme l'est une grappe. En effet, aucune de ces solutions n'aborde le problème des performances des E/S d'un point de vue global à l'ensemble des applications en cours d'exécution et la mise en concurrence des différentes optimisations engendre de nouveau une utilisation inefficace du système de stockage.

Après avoir introduit les nombreuses difficultés pour mettre en place un service d'optimisation global, nous avons proposé une stratégie à base de quantum. Cet algorithme a pour but principal l'amélioration des performances des E/S pour les modes d'accès particuliers aux applications *HPC* dans un contexte multi-applicatif. Cette stratégie a été intégrée à un service indépendant de contrôle, d'ordonnancement et de régulation nommé *aIOLi*. Plusieurs tests ont été conduits autour d'un serveur NFS utilisant le service offert par notre prototype. Les résultats obtenus sont très positifs, les gains sur certains jeux de tests sont notamment très significatifs. Le principal problème rencontré concerne les écritures séquentielles et la dépendance vis à vis de la politique de cache de type écriture retardée.

Dans ce chapitre, nous allons rappeler dans un premier temps les caractéristiques fondamentales de notre solution puis nous dresserons un bilan technique spécifique à la solution *aIOLi*. Par la suite, nous présenterons trois expériences qui ont été menées. Ces évaluations nous ont permis de découvrir certaines lacunes dans la mise en œuvre de notre proposition. De nouvelles optimisations permettant d'exploiter le service sur des architectures plus spécifiques sont en cours d'intégration [1] .

[1] Le projet s'est organisé autour d'une collaboration avec l'université de Czestochowa en Pologne depuis fin 2005 et plusieurs axes de recherches autour d'*aIOLi* sont en cours d'étude.

8.1. Le service *aIOLi*

Le service *aIOLi* est un module noyau *Linux*. Il a été mis en œuvre afin de proposer diverses stratégies d'ordonnancement pour des systèmes d'E/S. Dans cette section , nous effectuons un bilan général puis technique de l'état actuel du système.

8.1.1. Caractéristiques générales

Le service *aIOLi* peut être décrit de manière simplifiée comme suit : un système d'E/S, souhaitant utiliser le service, transmet ses requêtes en attente de traitement au sein des structures du module *aIOLi* (au lieu de les transmettre vers le support de traitement adéquat). Selon la stratégie d'ordonnancement en place, les requêtes vont être redistribuées, par la suite, au système d'E/S qui pourra alors les transmettre cette fois ci vers le support de traitement sous-jacent et retourner les réponses vers les processus concernés.

Plusieurs caractéristiques font de cette proposition une idée novatrice :

– Un service unique :
  Le système *aIOLi* est relativement simple du fait qu'il ne propose qu'un seul service, à savoir l'ordonnancement des requêtes. Certes, il a été conçu de manière à intégrer plusieurs stratégies d'ordonnancement qui peuvent être plus ou moins complexes mais il ne propose uniquement que ce type de service. Le traitement effectif des accès ou encore le déclenchement d'opérations de pré-chargement ou d'écriture retardée ont été volontairement laissés à la responsabilité des couches d'E/S du système de fichiers. Cette approche a été retenue afin de proposer un service générique indépendant des mécanismes spécifiques à chacun des systèmes. La cohérence des données ou les gestionnaires de caches ne sont pas altérés par le service.
  Par ailleurs, même si le système *aIOLi* a été conçu dans un premier temps pour la gestion des E/S disque, il est tout à fait possible de l'exploiter pour ordonnancer des requêtes réseaux ou tout autre type d'échanges pouvant être assimilés à des E/S. Dans ce dernier cas, il sera peut être nécessaire de modifier l'interface permettant de déposer les requêtes.
– Une indépendance vis à vis des applications :
  La seconde caractéristique fondamentale de notre proposition se situe au niveau de la notion de client. Dans la plupart des solutions, les clients sont les applications qui vont directement interagir avec une couche permettant d'optimiser leurs accès. Dans notre proposition, les applications n'ont et ne peuvent pas avoir connaissance du service *aIOLi*. Les «clients» sont les systèmes d'E/S qui viennent déposer les demandes «entrantes» en provenance des applications au sein de notre système. Cet aspect permet aux applications de profiter des optimisations sans pour autant recourir à une interface spécifique souvent fastidieuse à prendre en main. Cette richesse d'interface qui à première vue semble être intéressante dans les bibliothèques comme *ROMIO* est un des aspects qui n'a pas permis une large utilisation de ce genre de solution et les routines standard `POSIX` restent encore amplement utilisées.
– Une vue globale des interactions :
  Le service *aIOLi* a été conçu pour recevoir plusieurs requêtes en provenance de plusieurs clients (les systèmes d'E/S) pour une même unité de stockage. Ainsi, les stratégies d'ordonnancement ont une vision globale des interactions à destination du périphérique et peuvent réordonnancer les requêtes en conséquence. Dans le cadre d'une architecture distribuée, si le concept reste le même, l'interconnexion du service peut s'avérer un peu

plus délicate. Certes, l'association d'un système de fichiers centralisé comme un serveur `NFS` avec le module reste relativement simple : l'ensemble des accès transitant par les démons `nfsd`, ces derniers ont simplement à rediriger les demandes vers le service. Dans le cadre d'un système de fichiers parallèles où plusieurs nœuds sont exploités pour fournir l'architecture de stockage, il n'existe pas toujours un unique point de centralisation. Au contraire, les parties clientes du système de fichiers intègrent plusieurs mécanismes permettant d'accèder directement à un serveur d'E/S particulier. Dans ce cas, il est nécessaire de mettre en place une structure hiérarchique au sein de notre modèle : chaque point de la hiérarchie étant en charge par exemple d'un fichier (cf. section 6.1.1).
Dans tous les cas, l'idée principale est de s'appuyer sur une connaissance globale des requêtes en attente de traitement afin de fournir un ordonnancement optimisant les critères souhaités.

– Des stratégies d'ordonnancement «haut niveau» :
A la différence des politiques d'ordonnancement «de plus bas niveau» qui ont une vue restreinte de l'information, les stratégies pouvant être intégrées dans le support *aIOLi* reposent sur des paramètres de plus «haut niveau». A titre d'exemple, la variante de l'algorithme *MLF* proposée afin d'optimiser les E/S dans un contexte multi-applicatif *HPC* sélectionne les requêtes à traiter en s'appuyant sur les identifiants des fichiers, le type, les tailles et les *offsets*. Il en résulte une politique d'ordonnancement plus fine (le choix des paramètres sur lequel repose la stratégie étant plus large).
D'une manière générale, il serait tout à fait possible de rendre la fonction d'insertion (`aioli_add_request`) encore plus générique afin d'ordonnancer à différents niveaux. L'idée consisterait à fournir une structure particulière selon le système d'E/S permettant au service de trier les requêtes dans un premier temps. Par la suite, le contôleur d'E/S associé au client (le système d'E/S) exploiterait l'algorithme d'ordonnancement mis en œuvre pour ce type de structure. Une telle approche offre la possibilité de traiter tout type de requête d'E/S à différents niveaux : proche des applications, les algorithmes d'ordonnancement s'appuyeront alors sur les abstractions exploitées par les applications dans notre cas les requêtes fichiers ou bien les messages `MPI` ou au contraire à plus bas niveaux en travaillant sur les blocs ou les paquets réseaux.

L'élaboration du service *aIOLi* nous a amené à proposer un module noyau générique pouvant être interconnecté avec une large gamme de systèmes d'E/S. Si globalement, le service permet d'obtenir des performances significatives, plusieurs aspects doivent être finalisés (voire améliorés) afin de proposer un service totalement fiable pouvant être exploité en production. Nous abordons plusieurs de ces points dans le paragraphe suivant.

8.1.2. Bilan technique

Dans cette section, nous allons aborder plusieurs aspects techniques de la solution. Tout d'abord un rapide résumé de l'état actuel de la solution va être réalisé. Les modifications en cours et celles à venir sont décrites par la suite.

La solution *aIOLi* est un module noyau *Linux*. Les structures utilisées sont générales et le module doit pouvoir compiler avec la plupart des versions 2.6 du noyau *Linux*. Les dépendances avec les versions sont induites par les systèmes d'E/S. En effet, si l'interface d'interaction entre un client (le système d'E/S) et le service *aIOLi* est relativement simple (cf. section 6.2.2.5), l'interconnexion reste une étape intrusive et délicate puisqu'il est nécessaire d'instrumenter le code du client afin de détourner les requêtes vers le système *aIOLi*. Nous n'avons

```
                                avg
                     reqs     sum / aggs    bigg   last   shift   better
file id : c6174473
            read :     0        0 / 0        0      0      0        0
           write :   32771   26914 / 1553   1635    68      7       10
file id : c61ee577
            read :   32768   32699 / 3593    11     2      0        0
           write :     0        0 / 0        0      0      0        0
file id : c61c34ce
            read :   65547   65090 / 1247    133    128    107      167
           write :     0         0/ 0        0      0      0        0
file id : c6142c47
            read :     0        0 / 0        0      0      0        0
           write :   65536   62566 / 4366    262    245    153      887
file id : c61ad783
            read :   1026     1015 / 108     11     2      0        0
           write :     0        0 / 0        0      0      0        0
file id : c61e6a93
            read :     0        0 / 0        0      0      0        0
           write :   1024     1022 / 102     11     1      0        0
file id : c621fda3
            read :    129      126 / 13      11     1      0        0
           write :     0        0 / 0        0      0      0        0
                               ...
```

FIG. 8.1 – Copie d'écran du fichier `/proc/aioli/stats`

pas trouvé à ce jour une solution plus simple. Cependant cette opération doit être effectuée une seule fois et par un simple test, il est possible d'activer ou non l'utilisation du service *aIOLi* pour le système d'E/S.

A ce jour le système *aIOLi* a été interconnecté avec le système de fichiers `NFS` et également avec la couche `VFS`. Si la partie `VFS` reste à l'état expérimental, deux paquets combinant le serveur `NFS` et le service *aIOLi* peuvent être téléchargés sur le site du projet `http://aioli.imag.fr` (pour la version 2.6.16 et pour la version 2.6.17). Par ailleurs, plusieurs images *kadeploy* [2] permettant d'installer un serveur `NFS`-*aIOLi* (depuis les noyaux 2.6.12 jusqu'au 2.6.15) sont également disponibles sur la grappe de Sophia-Antipolis.

Ces paquets peuvent être exploités comme tels, l'intégrité des données n'étant pas mise en jeu et la quasi-totalité des fonctionnalités étant abouties. A titre d'exemple, la figure 8.1 correspond à une copie d'écran d'une partie du fichier `/proc/aioli/stats` lors de l'expérimentation incluant dix applications. Les informations sont récupérées puis inscrites par le sous-module de statistique décrit dans la section 6.2.2.4.

Pour chaque fichier, un identifiant est associé. Les informations disponibles sont triées selon le type de requête (lecture ou écriture). Pour chacun des types, les informations apparaissant correspondent au nombre de requêtes traitées, à la taille des requêtes en moyenne (nombre de requêtes agrégées divisé par le nombre d'opérations ayant généré des requêtes «virtuelles»).

[2] Outil de déploiement utilisé sur l'architecture Grid 5000 `https://www.grid5000.fr/`

Les valeurs qui suivent nous renseignent sur la taille de la plus grosse requête «virtuelle» traitée depuis l'ouverture du fichier ainsi que la taille de la dernière requête traitée. Les derniers champs sont propres à la stratégie d'ordonnancement mise en place. Dans l'algorithme que nous avons présenté, ces champs correspondent aux délais volontairement mis en place pour corriger soit des phénomènes de type décalage (cf. section 5.3.2) soit la possibilité de réaliser de meilleures agrégations. Les délais sont respectivement pour les décalages (*shift*) de 3 ms et pour les agrégations de 2 ms. Les temps ont été fixés suite aux nombreuses expérimentations.

Le processus actuel d'agrégation est relativement standard et pourrait être optimisé afin de continuer à améliorer encore les performances. Tout d'abord, la stratégie d'ordonnancement n'exploite pas la notion de degré de continuité (cf. annexe A). Les requêtes sont agrégées si elles sont contiguës. Il est prévu d'améliorer ce point afin de continuer à minimiser les déplacements. La définition d'un degré de continuité au sein de la stratégie d'ordonnancement va permettre de réduire par exemple les phénomènes de décalage. L'idée est la suivante : à la place de ce délai actuellement mis en place, une requête «d'anticipation» va être réalisée (en tenant compte de la taille de la portion manquante) et agrégée au sein de la requête «virtuelle» générée par l'algorithme d'ordonnancement. Ainsi, lorsque la requête va réellement arriver sur le serveur `NFS`, elle pourra être immédiatement servie par le cache. Ce mécanisme est en cours d'étude afin de savoir s'il a un impact sur la cohérence et l'intégrité des données.

Parallèlement, lorsque le processus d'agrégation est terminé, il n'est pas possible d'insérer un nouvel accès au sein de la liste représentant la requête «virtuelle» en cours de traitement. Ainsi, les accès qui arrivent en retard et qui pourraient être intégrés dans la liste, vont être traités par la suite, entraînant potentiellement des déplacements et une utilisation inefficace du système de stockage. Nous avons également prévu de modifier l'insertion pour pallier ce problème. La principale difficulté est de continuer à garantir les contraintes d'équité et d'interactivité entre l'ensemble des applications.

Enfin, nous rappelons que le maintien de la cohérence est assuré en s'appuyant sur l'estampille de chacune des requêtes et de leur type : si une écriture est en attente d'exécution pour un fichier donné, toute lecture sur ce même fichier avec une estampille plus récente ne pourra être retenue par l'algorithme d'ordonnancement. Une technique analysant le recouvrement des accès [3] pourra dans un second temps être également mise en œuvre.

Dans la suite de ce chapitre, nous allons décrire plusieurs expérimentations en cours. Ces évaluations ont permis de soulever plusieurs contraintes et c'est pour cela que nous avons choisi de les présenter.

8.2. Évaluations en cours et extensions

Les évaluations conduites dans le chapitre précédent ont toutes été réalisées sous les mêmes conditions et sur la même grappe : plusieurs processus accèdent à un serveur `NFS` exportant une partition `ext3` d'un disque *IDE*, chacun des processus étant déployé sur un nœud distinct afin d'éviter tout problème de concurrence.

Dans cette section, nous présentons trois expériences qui ont été menées en modifiant certains paramètres. Ces différents tests nous ont permis de révéler certaines lacunes dans la mise en place d'un serveur `NFS`-*aIOLi*. La première a été conduite sur la grappe de Sophia-Antipolis

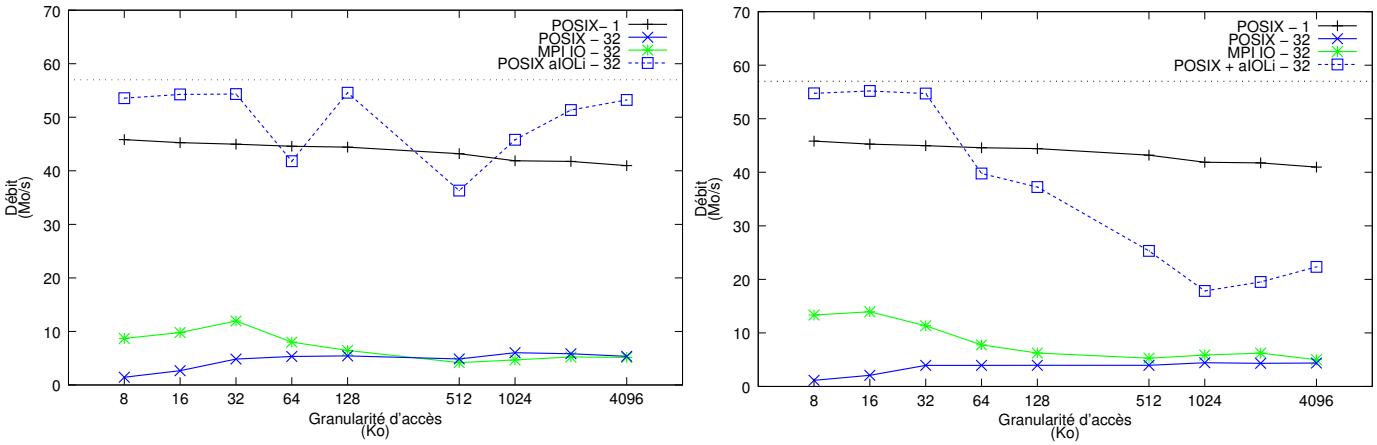
[3] *Byte Range Locking*.

afin d'étudier l'impact de la congestion du service NFS côté client. Les deux dernières ont été réalisées sur la grappe *icluster2* située dans les locaux de l'INRIA Rhônes alpes. Elles ont eu pour but d'analyser un système NFS-*aIOLi* au-dessus d'un disque *SCSI* puis au-dessus d'une baie *RAID* composé de 8 disques *SCSI*.

8.2.1. Le système NFS-*aIOLi* et les nœuds multi-processeurs

Lors de l'étude préliminaire, nous avons mentionné les problèmes de famine pouvant survenir au niveau des nœuds de calcul multi-processeurs [4] selon la couche cliente du service NFS (cf. section 4.1.3). Nous avons réexecuté le jeu de tests *IOR* afin d'évaluer une décomposition de 4 Go sur 32 processus déployés sur 16 nœuds (2 instances MPI par nœuds). La grappe de Sophia-Antipolis est constituée de nœuds bi-processeurs. La configuration du serveur NFS et les options de montage des clients sont identiques aux précédentes expériences en lecture.

La figure 8.2 permet de comparer les deux configurations et révèle l'impact dû à la congestion côté client.



32 processus déployés sur 32 nœuds — 32 processus déployés sur 16 nœuds

FIG. 8.2 – Décomposition d'un fichier de 4 Go - Lecture - Impact de la congestion côté serveur

Même si les performances fournies par l'approche «POSIX + aIOLi» restent meilleures que celles mesurées pour l'interface POSIX ou MPI I/O au-dessus d'un serveur traditionnel (figure de droite), nous pouvons constater une forte dégradation des performances à partir de 32 Ko. Comme nous l'avions indiqué, à partir de cette granularité, la stratégie d'ordonnancement utilisée dans le service *aIOLi* n'effectue plus d'agrégations et exploite seulement l'approche à base de fenêtres dédiées. Dans ce cas, plus la granularité d'accès est importante plus le système est bénéfique (ce qui peut être vu sur la figure de gauche).

Dans le cas où plusieurs processus s'exécutent sur un même nœud et du fait que chaque requête applicative de taille supérieure à 32 Ko est divisée en sous-requêtes NFS, un déséquilibre peut apparaître dans le traitement des accès (cf. section 4.1.3). Ainsi, le service *aIOLi* ne reçoit que certaines des requêtes pour une même période ce qui limite l'exploitation des fenêtres «dédiées» et donc des performances. Plus la granularité d'accès devient importante, plus la stratégie d'ordonnancement subit le phénomène de congestion de la partie cliente. Enfin, les accès entre les deux processus étant désordonnés, les techniques de pré-chargement ne sont pas exploitées du côté client ce qui défavorise également les gains de performance.

[4] Le problème peut également survenir dans les environnements multi-tâches.

L'approche en cours d'étude consiste à mettre en place un service de régulation au niveau de la *VFS* de chaque nœud. L'envoi des requêtes au serveur `NFS`-*aIOLi* sera réalisé de manière à optimiser les critères de performance en tenant compte des contraintes d'équité et d'interactivité. Cependant, comme nous l'avons indiqué, cette première temporisation peut avoir un impact sur la cohérence des données entre deux nœuds. Nous avons émis la possibilité de mettre en place une solution hiérarchique basée sur les fichiers. Le service *aIOLi* est déployé sur chacun des nœuds. Lorsqu'un processus accède au fichier, la première requête est interceptée par le service local qui demande au service principal de régulation si un nœud est déjà en charge de réguler les accès pour ce fichier. Si c'est le cas l'ensemble des accès sur ce fichier est transmis au site responsable sinon le service local devient le nœud en charge de la régulation pour ce fichier. Chaque service responsable d'un fichier transmet la requête «virtuelle» au site principal. Ce dernier met en place la qualité de service désirée entre les différents fichiers. Cependant, une telle approche requiert un nombre important de messages afin de transmettre et de faire suivre les demandes vers les différentes entités de régulation et nous craignons de retomber sur un problème de surcoût lié à la centralisation des messages comme cela a été le cas dans la première mise en œuvre de la stratégie (*aIOLimaster*, cf. section 6.1).

Les deux expériences suivantes se sont déroulés sur la grappe *icluster2* située sur Grenoble. Elles nous ont permis d'évaluer un système `NFS`-*aIOLi* au-dessus de disques *SCSI* dans un premier temps puis au-dessus d'une baie *RAID* par la suite.

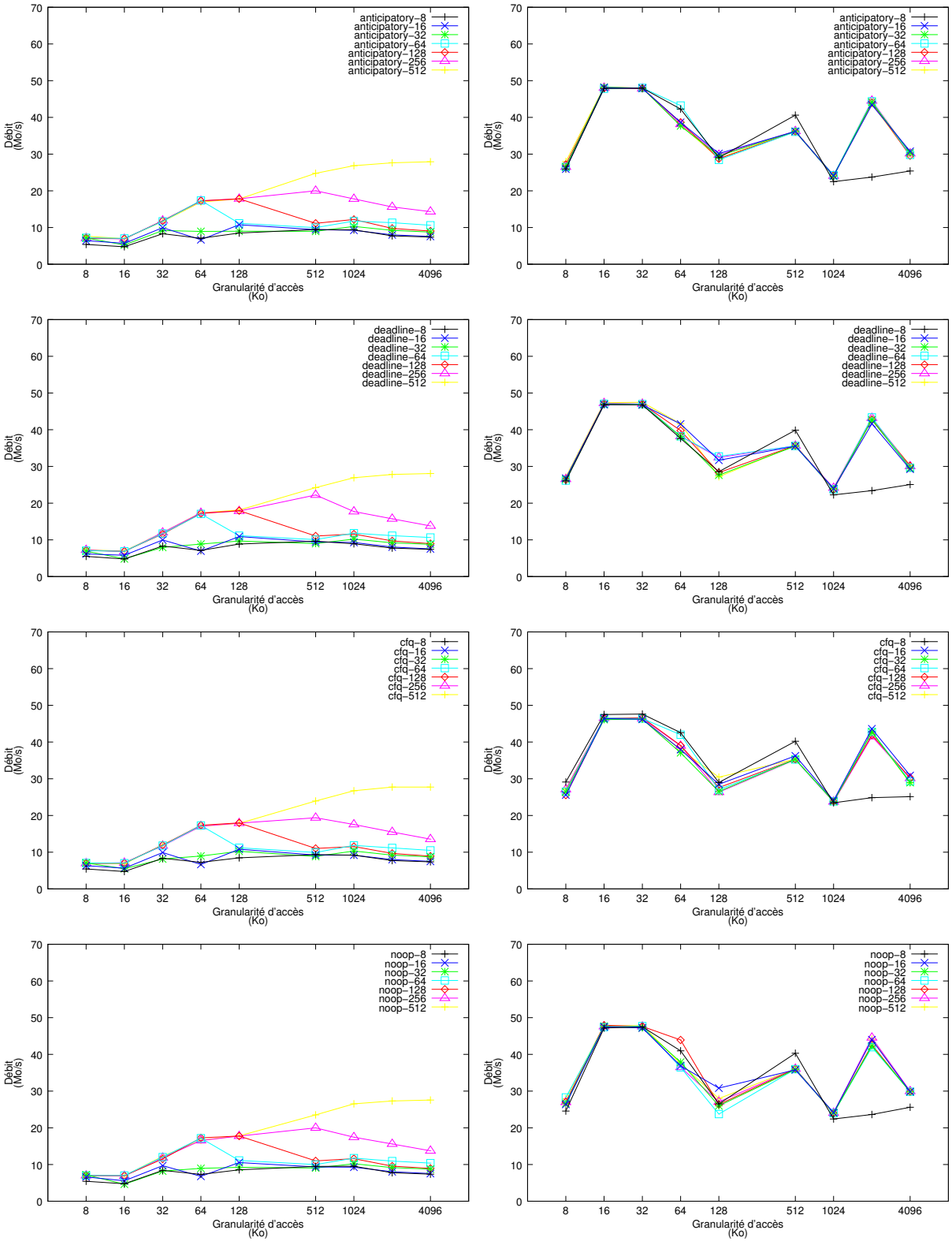
8.2.2. Le service *aIOLi* et les disques `SCSI`

Dans le cadre des expérimentations menées pour la validation de notre service, nous avons débuté une série de tests au-dessus d'un disque SCSI. L'architecture retenue a été la grappe *icluster2* [5] . Elle est composé de 104 nœuds interconnectés par un réseau Myrinet et par un réseau Giga Ethernet. Chaque nœud comprend deux processeurs *Itanium Mc Kinley* cadencés à 900 Mhz, 3 Go de mémoire vive et deux disques *SCSI* de 36 Go chacun. Le modèle utilisé pendant l'évaluation est un disque *SCSI Fujitsu* tournant à 10000 tours/min (modèle MAN3367MC). Le débit mesuré *via la commande* `hdparm` est d'environ 52 Mo. Pour l'expérimentation, 33 nœuds ont été réservés : un nœud a joué le rôle de serveur `NFS` et `NFS`-*aIOLi*, les 32 autres ont constitué les clients. Les paramètres liés au service `NFS` sont identiques à ceux utilisés sur la grappe de Sophia-Antipolis. Le jeu de test *IOR* a été de nouveau exécuté pour analyser une décomposition de 6 Go (2 fois l'espace mémoire disponible). Le test que nous présentons ici correspond à l'évaluation de l'impact du nombre de *thread* `nfsd` et des ordonnanceurs bas niveau. Ce test est relativement complet. Dans un premier temps, il permet d'étudier d'une part l'impact des différents paramètres système sur un serveur traditionnel exportant un partition d'un disque *SCSI*. Dans un second temps, la comparaison des résultats avec un serveur `NFS`-*aIOLi* va nous permettre d'analyser l'apport de la stratégie d'ordonnancement présente au sein du service *aIOLi*.

La figure 8.3 présente les différents graphiques.

En ce qui concerne le serveur traditionnel et en comparaison avec les valeurs mesurées par les disques *IDE* de Sophia-Antipolis (cf. section 7.2.2), les performances sont plus mauvaises. Par ailleurs, quelque soit l'ordonnanceur bas niveau sélectionné les performances sont similaires. La stratégie d'ordonnancement interne au disque semble avoir un important impact sur

[5] `http://i-cluster2.inrialpes.fr/`



NFS NFS-*aIOLi*

FIG. 8.3 – Impact de la sérialisation et apport du nombre de `threads` sur des disques `SCSI`
Evaluation des performances pour une décomposition en lecture de 6 Go par 32 processus répartis sur 32 nœuds. Les architectures `NFS` (à gauche) et `NFS`-*aIOLi* au-dessus d'un disque *SCSI* sont comparées.

les performances. Enfin, même si le nombre de *threads* améliore les performances, les valeurs atteignent à peine la moitié des capacités pouvant être fournies par le disque.

L'utilisation du service *aIOLi* est bénéfique puisqu'il permet au pire des cas d'obtenir au moins 50 % des capacités et ce quelque soit le nombre de *threads* `nfsd`. Cependant, en comparaison avec les améliorations apportées au-dessus d'un disque *IDE*, les gains sont limités. Globalement et même si les performances sont meilleures sur l'architecture `NFS`-*aIOLi*, il semble nécessaire de réeffectuer une série de tests en désactivant le mécanisme de queues présent dans le disque *SCSI* (cf. section 1.1.4) ou en initialisant la profondeur de la file à une taille inférieure. En effet, l'interaction de l'ordonnanceur bas niveau avec les mécanismes dit de *Queuing* semble se révéler rapidement inefficace [WNH01].

Dans la dernière expérience, nous présentons l'évaluation des performances d'une baie *RAID* composé de 8 disques *SCSI*.

8.2.3. Ordonnancement des requêtes dans la couche `RAID` logiciel

Même si comme nous l'avons mentionné, un grand nombre de grappes exploite encore un système `NFS` pour accèder aux données depuis les nœuds de calcul, la plupart repose sur un serveur spécifique exploitant généralement une baie de stockage (*RAID* 1+0 ou *RAID* 5+0).

Afin de valider notre approche, nous avons souhaité évaluer un système `NFS`-*aIOLi* dans des conditions similaires. Nous avons exploité une baie de stockage composé de 8 disques *SCSI* de 140 Go (Seagate STI3146807LC) disponible sur l'architecture *icluster2*. La baie et le serveur *ita102* ont été dédiés à nos expériences pendant une quinzaine de jours afin de ne pas parasiter les résultats par d'éventuelles applications concurrentes. L'outil `mdadm` [6] a été utilisé afin d'abstraire les 8 disques en *RAID* 0. La taille de répartition (*chunk*) a été fixée à 64 Ko. La commande `hdparm` nous a permis de mesurer un débit maximum à 188 Mo/s.

L'expérience qui a consisté à analyser une décomposition d'un fichier de 6 Go a été réalisée au-dessus de 32 processus déployés sur 32 nœuds *via* le jeu de tests *IOR*. Comme pour la plupart des tests, nous avons fait varier la granularité d'accès de 8 Ko à 4 Mo. Les résultats apparaissent à la figure 8.4.

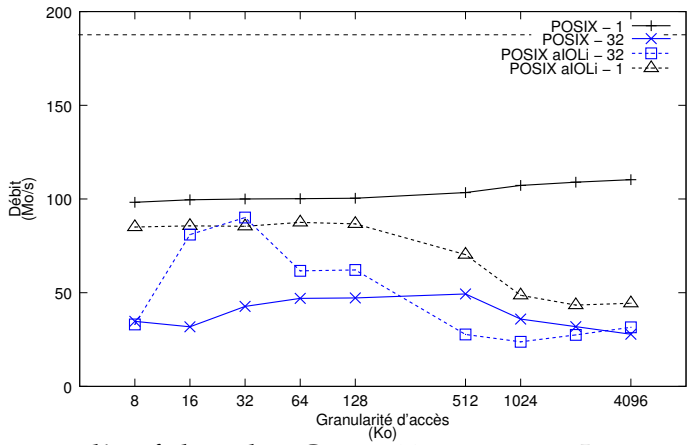


FIG. 8.4 – Décomposition d'un fichier de 6 Go sur 32 processus - Lecture - `NFS` *vs* `NFS`-*aIOLi*
Comparaison entre les performances fournies par les routines `POSIX` sur un serveur NFS et un serveur `NFS`-*aIOLi* au-dessus d'une baie *RAID* 0 composée de 8 disques *SCSI*

[6] *Multiple Devices ADMin*.

Dans un premier temps, nous allons anlayser les performances délivrées par l'interface POSIX en séquentiel puis en parallèle. En observant la courbe «POSIX - 1», nous pouvons constater que l'approche séquentielle ne permet pas d'atteindre des performances proches du débit maximum. Si, pour un disque *IDE*, les performances lors de tels modes d'accès atteignent environ 80% (cf. section 4.2), dans le cas présent, les valeurs mesurées s'élèvent au mieux à 58% du débit maximum (courbe en pointillé). Lorsque la granularité est inférieure à 64 Ko, la parallélisation des accès est limitée, la taille de répartition ayant été fixée également à cette valeur. Pour les granularités supérieures, deux facteurs doivent être pris en compte. Certes les accès peuvent être réalisés en parallèle : une requête de 128 Ko est répartie entre deux unités de stockage. Cependant, la granularité du protocole du système de fichiers (ici, NFS 32 Ko) transmet des requêtes qui peuvent être traitées dans un ordre inefficace : chaque requête NFS est réceptionnée par un *thread* nfsd puis transmise à la pile des E/S. La couche d'abstraction des disques [7] redirige les sous-accès vers les queues de l'ordonnanceur bas niveau associé à chacun des disques. Comme pour les disques *IDE* (cf. section 7.2.1.1), l'ordre de dépôt des requêtes NFS dans la couche d'abstraction des disques n'est pas strict. Ainsi, même si les performances sont légèrement meilleures pour les grosses granularités qui profitent du *RAID* 0, les performances sont limitées.

La courbe «POSIX - 32» corrobore ces remarques. En effet, quelque soit la granularité le parallèlisme des accès sur les disques doit être exploité. Par exemple, pour une granularité de 8 Ko, chacun des processus génère une requête NFS de 32 Ko, soit 1 Mo de données à traiter. Ces données peuvent impliquer dans le pire des cas un seul disque et dans le meilleur la totalité de la baie [8] . Dans ce dernier cas, $2 * 64$ Ko doivent être fournis par chacun des 8 disques. Les accès sont donc traités en parallèle sur l'ensemble de la baie. Cependant, à la vue des performances atteintes, l'utilisation du *RAID* 0 n'est pas suffisante et une stratégie d'ordonnancement définissant un ordre strict entre les accès semble requise.

La variante *MLF*, qui a été conçue pour ce but dans le service *aIOLi*, sérialise les accès en considérant l'unité virtuelle abstrayant les 8 disques (/dev/md0) comme un unique disque. Ainsi, losrque le processus d'agrégation génère une requête «virtuelle» de taille conséquente, chacun des sous-accès de 32 Ko est réalisé l'un après l'autre et les performances susceptibles d'être apportées par le *RAID* 0 ne sont pas exploitées. Il en est de même pour l'approche séquentielle au-dessus de l'architecture NFS-*aIOLi* (courbe «POSIX + *aIOLi* - 1»).

Par ailleurs, en comparant avec la forme globale des courbes obtenues lors de l'étude d'une décomposition par 32 processus d'un fichier de 6 Go stocké sur un disque *SCSI*, nous pouvons constater une forte similitude. Les mécanismes internes aux disques *SCSI* semblent avoir également un rôle important dans les performances. Comme nous l'avons dit au paragraphe précédent, il est important de réaliser l'ensemble des tests en désactivant les techniques de *Queuing* présentes dans ces unités de stockage.

Afin d'exploiter efficacement le parallélisme au sein du service *aIOLi*, une première solution pourrait consister à mettre en œuvre une approche «pipelinée» comme cela a été réalisée pour réduire les délais lors des écritures (cf. section 6.2.4). Cette solution qui peut être activée par un simple test risque, toutefois, d'avoir un impact sur les techniques de pré-chargement exploitées par la pile d'E/S côte serveur : chaque sous-accès qui compose la requête «virtuelle» va être déposé de manière parallèle dans la pile d'E/S cassant ainsi l'ordre strict favorable aux performances.

[7] Pour rappel, elle se situe au niveau de la couche «bloc», /dev/md0 (cf. section 1.2.2.3).

[8] Nous rappelons que le décalage entre les processus MPI a un rôle important dans l'enchevêtrement des requêtes.

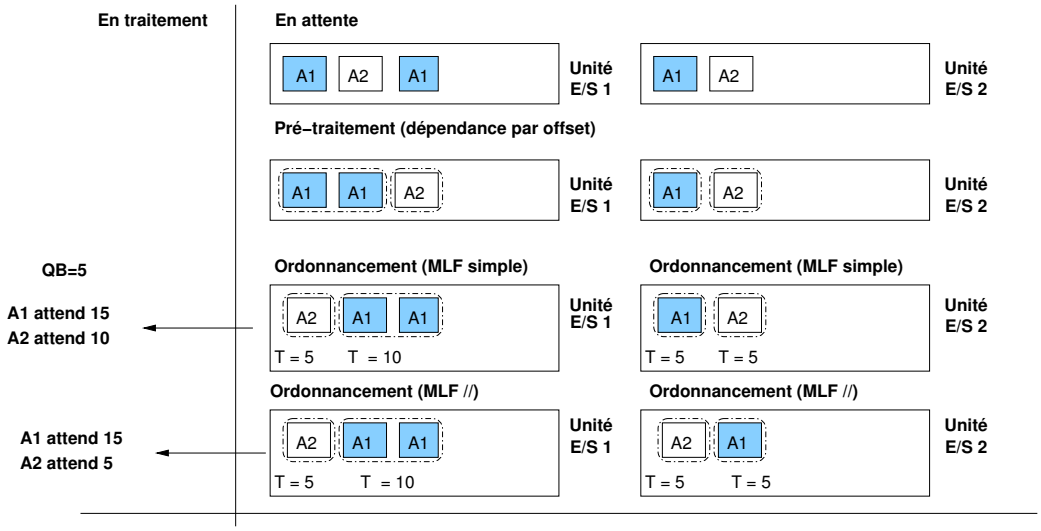


FIG. 8.5 – Prise en compte de la répartition des données pour une baie *RAID*

Une seconde méthode consiste à exploiter les informations de répartition au sein de notre algorithme d'ordonnancement. Le service *aIOLi* a la possibilité de récupérer les informations relatives à la stratégie *RAID* mise en place. En s'appuyant sur ces informations, les accès sont sérialisés par unité de stockage et non plus en considérant les 8 disques comme un et seul même support. Une telle stratégie va permettre de prendre en compte les dépendances entre les applications et continuer à améliorer les performances globales du système. La figure 8.5 reprend l'exemple présenté dans la section 5.1.2. Il permet d'illustrer la dépendance entre les applications lors du choix d'un ordonnancement lorsque les données sont distribuées sur plusieurs serveurs d'E/S (ou unités de stockage dans le cas d'une baie *RAID*).

Une queue d'ordonnancement est associée à chacune des unités. La variante *MLF* est appliquée sur chacune des files d'attente (processus d'agrégation, élection des requêtes). Parmi les requêtes pouvant être sélectionnées dans chacune des queues, un ordonnancement global permet de déterminer l'ordre le plus opportun. Dans l'exemple illustré,une première application a émis deux requêtes à destination du disque 1 et une requête à destination du disque 2. Un second programme a emis deux requêtes impliquant chacune un disque. Le processus d'agrégation est réalisé et l'algorithme d'ordonnancement est appliqué. Si l'étape permettant d'appliquer une vision globale n'est pas réalisée, l'application 2 est défavorisée de manière inutile. Dans l'autre cas, l'application 1 ne requiert pas plus de temps et l'application 2 bénéficie d'une réduction de son temps de complétion.

Pour déterminer les requêtes qui doivent être ordonnancées, la «vision globale» consiste à appliquer de nouveau notre variante *MLF*. L'algorithme reste identique ; seul le concept de requête «virtuelle» change légèrement : une requête virtuelle est constituée des sous-accès dépendants entre eux. Ainsi, si une requête implique plusieurs unités de stockage, l'ensemble de ces sous-requêtes forme une requête virtuelle. A ce jour, il a été planifié d'intégrer ces derniers aspects dans le prototype *aIOLi*. Ce travail est une perspective supplémentaire à moyen terme autour des travaux menés en collaboration avec l'université de Czestochowa en Pologne.

8.3. Vers une solution de production

Nous avons rappelé les caractéristiques qui font de la solution *aIOLi*, un système novateur. Sans recourir à une interface spécifique, les E/S générées par les applications *HPC* sont contrôlées, ordonnancées et régulées afin d'améliorer les performances tout en maintenant une qualité de service. Les performances obtenues lors d'une première série de tests sur une configuration constituée d'un serveur `NFS` exploitant un disque *IDE* ont montré l'apport de notre proposition en proposant des améliorations significatives notamment pour les opérations de type lecture (cf. chapitre 7). La version téléchargeable sur le site du projet est aboutie pour ce type de configuration et les améliorations planifiées à moyen terme se concentrent principalement sur le processus d'agrégation interne à la stratégie d'ordonnancement. Ces dernières optimisations vont continuer à minimiser les déplacements de têtes de lecture/écriture et à maximiser l'exploitation des techniques de pré-chargement.

Dans le but de proposer une solution `NFS`-*aIOLi* de production susceptible d'être mise en œuvre sur n'importe quel type de configuration, nous avons entamé une seconde campagne de tests. Ces nouvelles expériences se concentrent sur les disques *SCSI*. Nous souhaitons comprendre comment une stratégie d'ordonnancement «haut niveau» se combine aux mécanismes internes à ce type de support. Même si l'exploitation du service *aIOLi* permet d'améliorer les performances lors de comportements parallèles, les premières expériences nous ont révélé des gains moins significatifs que pour les disques *IDE*. Nous avons planifié de réaliser de nouveaux ces tests en désactivant les techniques de *Queuing*. En effet, ce type de mécanisme peut conduire à un traitement inefficace des sous-accès.

Parallèlement, nous avons commencé à étudier l'utilisation du service *aIOLi* sur une baie de stockage. Faute de matériel, nous n'avons pas pu analyser les performances sur une baie à base de disques *IDE*. Les tests ont été réalisées sur une architecture *RAID* 0 à base de 8 disques *SCSI*. Ces expériences nous ont permis de révéler l'importance d'ajouter au sein de notre algorithme d'ordonnancement la prise en compte de la répartition des accès sur les différents disques. La mise en place de ces aspects au sein du service est actuellement en cours, le but étant de proposer une sérialisation au niveau de chaque unité de stockage. Dans un premier temps, les paramètres du *RAID* seront fournis lors de l'interconnexion du système d'E/S afin de faciliter l'évaluation. Par la suite, il a été prévu que le service *aIOLi* interagissent directement avec les sous-couches de la pile d'E/S.

La finalisation de ces aspects va permettre de proposer un serveur `NFS`-*aIOLi* adapté à la plupart des configurations exploitées sur les grappes de petites et de moyennes tailles. Ces travaux entrent dans le cadre de la collaboration mise en place depuis fin 2005 avec une université polonaise.

Le projet *NFSp* [9] [Lom03] a été porté sous le noyau 2.6 et nous sert à présent de base pour les nouveaux tests et la mise en œuvre des contraintes liées à la gestion parallèle des accès au sein du service *aIOLi*.

Une des principales difficultés dans la mise en place du service *aIOLi* au sein d'un système de stockage parallélisé réside dans la centralisation des accès nécessaires pour appliquer une stratégie d'ordonnancement global. Les approches à base de *RAID* logiciel ou à la *NFSp* reposent sur un point central et facilitent donc le contrôle, l'ordonnancement et la régulation

[9] Proposée en 2002, ce système a consisté à mettre en place un serveur `NFS` exploitant plusieurs serveurs d'E/S pour stocker les informations. Les clients ne communiquent qu'avec le serveur `NFS` qui redirige les requêtes vers les différents serveurs de stockage. Les accès sont parallélisés et les performances sont par conséquent meilleures, `http://nfsp.imag.fr/`.

des E/S. Dans les systèmes totalement distribués, la mise en place d'une solution hiérarchique semble être l'approche à étudier. Le point dé départ consiste à réguler les E/S par fichier puis d'appliquer une stratégie d'ordonnancement entre les différents points.

A plus long terme, il a été planifié d'étudier la mise en place de routines permettant une meilleure interaction avec les gestionnaires de caches présents tout au long de la pile d'E/S qui compose le système de fichiers. Le premier but est de pallier les problèmes liés au stratégie d'écriture retardée qui font défaut lors de comportements séquentiels en écriture. De même, il semble important de prévoir les stratégies asynchrones qui commencent également à être exploitées dans plusieurs codes. D'une manière globale, nous souhaiterions compléter la stratégie d'ordonnancement afin de transmettre directement aux clients (les systèmes d'E/S) les accès qui peuvent être satisfaits directement par le gestionnaire de cache. Cette approche va permettre d'améliorer les lectures et les écritures en supprimant les délais liés à la sérialisation. La difficulté réside dans l'endroit où les requêtes sont redirigées vers le service *aIOLi*.

Par ailleurs, une seconde stratégie [LDV05] a été proposée lors la mise en œuvre du service *aIOLimaster* (cf. section 6.1). Son étude a temporairement été délaissée afin de nous concentrer sur le développement de la solution *aIOLi*. Cet algorithme est une variante de la politique *Weighted SJF* qui minimise la somme des temps de réponse tout en évitant les problèmes de famine. Nous avons planifié de comparer cet algorithme avec la variante *MLF* actuelle.

Le service *aIOLi* a été mis en œuvre dans le but de proposer un support pour contrôler, ordonnancer et réguler les E/S disques. Les différents travaux conduits au cours de l'élaboration du service *aIOLi* nous ont amené à proposer un module noyau générique pouvant être interconnecté avec une large gamme de systèmes d'E/S. Même si les caractéristiques de ce module offrent encore de nombreuses possibilités d'évolution, les grandes lignes de son architecture en sont largement tracées.

Le prochain et dernier chapitre va conclure ce mémoire. Nous allons résumé les divers sujets abordés en rappelant l'implication de plus en plus importante des E/S au sein des nouvelles applications et l'impact sur les performances lors de modes d'accès parallèles. Par la suite, nous évoquerons plusieurs perspectives autour du projet et les autres pistes qui peuvent également contribuer à l'amélioration globale des performances des systèmes de stockage au sein des architectures dédiées au calcul intensif.

CONCLUSION

Rappel du contexte et des objectifs

La gestion des données est, depuis fort longtemps, une composante clé au sein d'un système informatique. Dépendant de l'architecture sous-jacente, les nombreuses couches qui composent les systèmes de stockage n'ont cessé de s'adapter dans le but de proposer des solutions permettant de pallier l'écart de performance entre les puissances d'analyse et les débits/temps d'accès fournis par les disques. Deux approches sont généralement mises en œuvre : la première solution «matérielle» consiste à paralléliser le traitement des E/S en exploitant plusieurs unités de stockage ou de serveurs d'E/S, la seconde repose sur la mise en place de techniques logicielles permettant de limiter les interactions avec les disques et le cas échéant les positionnements des têtes de lecture/écriture. Ces dernières sont des opérations extrêmement coûteuses en temps (9 ms en moyenne sur les unités actuelles), il est par conséquent impératif de les limiter. Ces aspects ont été abordés en début de manuscrit dans le chapitre 1.

Parallèlement aux considérations «bas niveau» énoncées ci-dessus, le développement croissant d'applications scientifiques distribuées, résultant du fort essor des architectures parallèles, a modifié la conception des systèmes de fichiers modernes. La prise en compte de nouveaux paramètres tels que la quantité d'informations manipulées ainsi que les modes d'accès mis en œuvre (accès concurrents à grains fins, multiplicité des accès parallèles disjoints, ...) est requise. Les requêtes générées par ce type d'application ne peuvent être gérées de manière efficace par les couches usuelles de la pile d'E/S qui n'ont pas été prévues pour de tels comportements, et les fortes dégradations que subissent les systèmes de stockage ont un réel impact sur les performances globales des applications.

Les solutions basées sur du *RAID* matériel ou logiciel permettent d'avoir des performances globalement meilleures. Malheureusement, ce genre d'approche a plusieurs limitations. Premièrement, chacune des unités n'est pas exploitée au maximum de ses capacités : le débit «agrégé» (ou cumulé) de l'ensemble des unités permet de satisfaire en partie les besoins de l'application mais à un ratio coût/performance relativement important. Par ailleurs, le découpage utilisé pour répartir les données est prépondérant dans les performances puisqu'il doit impérativement être calqué sur le découpage utilisé par l'application parallèle. D'une manière général, ces solutions contournent le problème induit par les modes d'accès spécifiques aux applications parallèles *HPC* au lieu d'essayer de le résoudre.

Les travaux qui se sont concentrés sur cette problématique ont souvent mis à défaut les méthodes courantes d'accès (`open`, `read`, `write`, `close`, ...) en leur imputant le fait qu'elles ne permettent pas, par exemple, d'accéder de manière optimale à des informations diffuses au sein d'une même ressource ou encore leur impossibilité à gérer de manière fine la correspondance entre les vues applicatives (les fichiers) et le placement des données physiques (les blocs disque).

Plusieurs approches «logicielles» permettant de corriger ces aspects ont été proposées, nous les avons synthétisées dans le chapitre 2.

La plupart de ces techniques ont été exploitées dans les systèmes dédiés à la gestion des E/S parallèles. Dans un premier temps, des systèmes de fichiers spécifiques ont été redéveloppés. Cependant, leur complexité d'utilisation ne leur ont pas permis d'être adoptés par la communauté. L'idée retenue a consisté à compléter les routines standard par le biais de bibliothèques fournissant des fonctionnalités propres à la thématique des E/S parallèles : principalement des fonctions permettant l'accès à des données diffuses mais répondant à un schéma structuré. Le chapitre 3 a décrit plusieurs de ces systèmes.

En 1997, le consortium MPI a défini une nouvelle interface dans le but de définir les divers prototypes d'E/S au sein des applications parallèles. Ce standard, communément appelé MPI I/O, fait office de référence aujourd'hui et l'ensemble des travaux suggérés par la communauté sont axés autour de ce paradigme.

Cependant, l'utilisation, habituellement recommandée par une large partie de la communauté scientifique, de bibliothèques mettant en œuvre ce standard nous semble discutable sur 2 axes :

- La richesse des interfaces :
  La complexité des routines définie comme principal inconvénient des systèmes de fichiers dédiés aux E/S parallèles peut également être imputée au standard MPI I/O. En effet, afin de tirer le meilleur gain d'une bibliothèque proposant les différents niveaux d'optimisation de MPI I/O, il est primordial d'avoir une profonde connaissance de l'ensemble des subtilités de chacune des fonctionnalités (une soixantaine de routines est disponible !).
- Les aspects mono-applicatifs :
  La problématique des E/S parallèles semble être traitée, par habitude, d'un point de vue mono-applicatif : une application composée d'un nombre plus ou moins important de processus accède de manière parallèle aux ressources de stockage. Certes la proposition de solutions pour améliorer les performances dans un tel cas est primordial. Cependant, cette position n'est pas appropriée aux environnements multi-applicatifs comme le sont la plupart des grappes, chacune des applications ayant un impact sur les autres. En d'autres termes, lorsqu'un programme utilise une bibliothèque d'E/S parallèles afin d'être plus efficace, l'ensemble des optimisations ne tient compte que des requêtes qui lui sont propres. De ce fait, les stratégies mises en œuvre pour optimiser les performances s'avèrent inutiles et coûteuses lorsqu'elles entrent en conflit avec d'autres requêtes provenant d'applications concurrentes au niveau des serveurs de stockage.

Dans un contexte de calcul intensif et d'architecture parallèle distribuée, le système de stockage, local ou distant, semble le seul capable de prendre en compte la totalité des interactions émanant de l'ensemble des applications s'exécutant sur l'architecture parallèle et de mettre en œuvre les méthodes d'optimisation les plus adaptées. Cependant, le nombre de systèmes de fichiers proposé par la communauté depuis ces vingts dernières années est considérable et il ne nous a pas semblé judicieux de redévelopper un système complet.

L'approche que nous avons retenue a consisté à étudier dans quelle mesure il est possible d'obtenir des performances en s'appuyant uniquement sur les routines POSIX, et, si cela est réalisable, comment une politique d'optimisation globale à l'ensemble des applications s'exécutant sur la grappe peut être mise en œuvre.

Solution proposée

L'étude préliminaire décrite au chapitre 4 a consisté à analyser les performances lors de modes d'accès parallèles dans un cadre mono puis multi-applicatif. Ce premier travail nous a permis de valider les remarques effectuées précédemment. L'exécution du jeu de tests *IOR* sous différentes configurations nous a révélé l'impact sur les performances pour une, deux et dix applications au-dessus d'un serveur `NFS`. Certes, s'il est vrai que le choix d'un serveur `NFS` peut être discuté pour des architectures à très grande échelle, il reste le service le plus déployé pour partager les données sur des grappes de petite et moyenne taille (inférieure à 200 nœuds). Dans un tel contexte, le serveur doit supporter un nombre conséquent d'accès simultanés, ce qui constitue un cadre tout à fait approprié pour évaluer l'intérêt d'un service d'ordonnancement global.

Parallèlement, les diverses expérimentations conduites nous ont amené à étudier les performances fournies par une approche sérialisée. Nous avons vu qu'il était possible, en s'appuyant sur ce mécanisme, de maintenir un niveau de performance plus élevé. La mise en place d'un point de centralisation permettant la régulation de l'ensemble des interactions des E/S nous a paru intéressante du fait de la possibilité d'appliquer une stratégie d'ordonnancement global susceptible d'apporter, d'une part, de meilleures performances en favorisant la contiguïté entre les accès et, d'autre part, une qualité de service entre les applications primordiale dans un contexte multi-applicatif.

En effet, une approche naïve pour maximiser les performances consisterait à traiter les accès programme par programme : l'ensemble des accès serait récupéré au sein du système de fichiers, les requêtes seraient réordonnées de manière à favoriser les techniques de préchargement et chaque fichier serait manipulé l'un après l'autre. Cela fournirait d'excellentes performances globales mais aurait un impact significatif sur l'interactivité en plus d'engendrer des problèmes de famine, critères prépondérants dans une architecture multi-applicative comme l'est une grappe.

La définition d'un algorithme d'ordonnancement global pour les E/S a été abordée dans le chapitre 5. Les différentes recherches conduites au sein de la thématique d'ordonnancement nous ont montré la difficulté de fournir un modèle précis recouvrant l'ensemble des contraintes relatives à notre contexte (modes d'accès parallèles, granularité applicative *vs* spécificité des systèmes de fichiers). Par ailleurs, nous avons évoqué l'impossibilité de garantir les trois critères que nous souhaitons optimiser, à savoir l'efficacité, l'équité et l'interactivité.

Ainsi, nous avons proposé une heuristique basée sur la stratégie d'ordonnancement *Multiple Level Feedback*. Ce type d'algorithme est souvent utilisé dans l'ordonnancement des processus. Il repose sur des quantas attribués aux différents processus. Dans notre variante, la tâche traitée (une requête ou un agrégat de requêtes) est sélectionnée en fonction d'un quantum dynamique qui varie selon le temps d'attente au sein de la queue. Le temps alloué est fonction d'un historique associé à chacun des fichiers manipulés. Cette approche à base de quantum permet de répartir l'accès à la ressource entre toutes les requêtes qui sont présentes dans le système. Par le biais d'échéances limites, une équité et une interactivité entre les applications sont garanties.

Les travaux réalisés autour de la mise en place d'une solution de contrôle, d'ordonnancement et de régulation des E/S ont été présentés dans le chapitre 6. Ils ont conduit à la mise en œuvre du service *aIOLi*. Ce système est un module noyau *Linux*. Il fournit aux différents

systèmes d'E/S un service d'ordonnancement totalement indépendant et générique. Les applications n'ont en aucun cas connaissance de ce service.

Un des points forts de ce système est qu'il a été conçu afin de faciliter l'intégration de nouvelles stratégies d'ordonnancement. Cette fonctionnalité permet de faire du système *aIOLi* un support d'évaluation d'algorithmes pour la gestion des E/S.

Le système de fichiers `NFS` a été instrumenté de manière à ce qu'il puisse exploiter les services proposés par la solution *aIOLi*. Les diverses expérimentations présentées au chapitre 7 ont permis de valider l'utilisation de notre proposition. Des gains considérables ont été mesurés dans plusieurs configurations et l'évaluation autour d'un jeu de tests composé de dix applications a montré sans équivoque l'intérêt d'un service de contrôle, d'ordonnancement et de régulation des requêtes au sein d'une architecture parallèle basée sur un serveur `NFS` pour partager les données.

Ces résultats ont permis de valider nos objectifs :

- Les applications exploitant des schémas d'accès parallèles atteignent des performances supérieures aux stratégies collectives présentes dans la bibliothèque *ROMIO* et ce en exploitant simplement les routines standard `POSIX`.
- La mise en place d'une politique d'ordonnancement global permet d'améliorer significativement les performances de chacune des applications s'exécutant sur la grappe.

Le chapitre 8 a dressé un bilan général des travaux conduits autour du projet *aIOLi* et a introduit les opérations actuellement en cours autour des disques *SCSI* et des solutions à base de parallélisme. Le but est de fournir avant la fin 2006, une version de production totalement aboutie. Une version pour les systèmes *NFS* au dessus des disques *IDE* est d'ores et déjà disponible sur le site `http://aioli.imag.fr`.

Depuis fin 2005, l'ensemble des travaux est réalisé en collaboration avec l'université de Czestochowa située en Pologne. Le but de ce partenariat est de proposer des techniques de gestion de données s'appuyant sur des briques fondamentales largement exploitées. La solution *aIOLi* qui repose uniquement sur l'interface `POSIX` et qui peut être interconnectée avec une large gamme de systèmes d'E/S entre tout à fait dans ces considérations.

Perspectives

D'un point de vue plus général, les caractéristiques du service *aIOLi* offrent encore de nombreuses possibilités d'évolution et plusieurs pistes ont été ouvertes lors du chapitre précédent.

Le principal défi va constituer à interconnecter notre proposition à un système totalement distribué où aucun point logiciel «central» n'est présent. La mise en place d'une hiérarchie apparaît comme étant la marche à suivre, nous l'avons abordée dans les chapitres 6 et 8. La mise en place d'une telle stratégie devrait pouvoir être facilitée par le concept de «méta-nœud» mis en œuvre dans plusieurs systèmes de fichiers. Dans de tels systèmes, chacun des nœuds clients devient tour à tour responsable du maintien de la cohérence pour un fichier : l'ordonnancement des accès pour ce fichier serait alors associé au maintien de la cohérence.

Cependant, même si la régulation des accès est répartie entre plusieurs nœuds, l'utilisation d'un site de régulation «maître» pour l'ensemble de la hiérarchie semble requise : une vision globale étant nécessaire pour appliquer les meilleurs choix. La mise en place d'un tel point «logiciel» a été partiellement évaluée lors de la proposition du service *aIOLimaster* et nous avons

pu observer plusieurs surcoûts induits par l'échange des messages nécessaires pour appliquer la sérialisation des accès. Une approche à base de prédiction permettant de recouvrir les délais peut être une solution. Cependant, le prototype réalisé n'a pas permis d'obtenir des résultats concluants. Le modèle utilisé pour calculer les prédictions n'était pas assez fin et la plupart des prédictions étaient erronées. Il serait intéressant d'étudier des modèles plus fins. Un point de départ pourrait consister à étudier les méthodes de prédiction basées sur des valeurs statistiques tenant compte de l'historique. L'outil *Network Weather Services* [Wol98] s'appuie sur de telles techniques pour prédire les performances réseaux.

Une autre possibilité afin de réduire ces surcoûts pourrait également consister à mettre le service au sein des points centraux «physiques» (à *contrario* des points «logiciels») tels que les commutateurs ou les passerelles réseaux. A ce jour, cette proposition n'est qu'une éventualité et n'a pas été approfondie. Cependant, en s'appuyant sur la thématique des réseaux actifs, il nous paraît intéressant d'étudier comment la gestion des E/S pourrait être combinée aux politiques de qualité de service réseau déjà présente dans de nombreux équipements. Dans une telle approche, chaque requête d'E/S est analysée au moment de son passage dans le matériel réseau et un service similaire à celui d'*aIOLi* pourrait appliquer diverses optimisations.

La gestion des E/S dans les architectures de type grappe et grappe de grappes se complexifie. Les concepts du partage de données autrefois étudiés de manière globale deviennent des axes de recherches à part entière (gestion de méta-informations, espace de nommage multi-sites global, tolérance aux pannes, ...). L'axe que nous avons abordé tout au long de ce manuscrit a été celui des performances. Nous avons choisi d'étudier l'ordonnancement des E/S pour l'ensemble de l'architecture afin d'exploiter efficacement les unités de stockage. La solution proposée s'appuie uniquement sur les routines standard `POSIX` pour des critères de simplicité. Une approche complémentaire pourrait consister à créer une couche permettant d'interconnecter le service *aIOLi* aux couches des bibliothèques `MPI I/O` le cas échéant. De ce fait, le service pourra bénéficier des «vues» (les schémas d'accès) définies par certaines des routines *MPI I/O* et continuer à affiner l'exploitation des systèmes de stockage. Par ailleurs, l'utilisation de ces vues pourrait être bénéfique aux gestionnaires de cache sous-jacents. En s'appuyant sur ces informations, des techniques de pré-chargement efficace pourraient être mises en place.

D'une manière générale, les E/S prennent de plus en plus d'importance dans la gestion des ressources des architectures dédiées au calcul intensif. A titre d'exemple, les aspects CPU souvent pris en compte lors de l'attribution ou la migration des tâches sur des nœuds sont complétés par les contraintes de stockage [OJM06]. Parallèlement, plusieurs travaux autour de la mise en place de nouveaux gestionnaires de ressources prenant en compte les quantités de données manipulées par les applications sont présentées [ZYSM03] [GGC+05]. Une thèse axée sur l'optimisation des ressources d'une grappe vient de débuter autour d'une collaboration entre le laboratoire ID-IMAG et la société BULL. La prise en compte des quantités de données manipulées par chacune des applications lors de la soumission des tâches sur l'architecture est l'une des facettes de cette thèse.

Globalement, les équipements physiques et les couches logicielles pour les nœuds destinés au calcul intensif nécessitent des mécanismes qui tiennent compte des dégradations induites par l'exécution concurrente de plusieurs applications. Il ne suffit plus d'essayer d'optimiser l'exploitation des performances CPU mais de tenter d'améliorer l'utilisation des performances des E/S disque ou encore de l'échange des messages réseaux. Ces derniers sont, comme les requêtes disque, assujettis à la concurrence multi-applicative [MM05].

La mise en place de techniques spécifiques visant à contrôler et optimiser l'exploitation de chacune des unités semble de notre point de vue une approche commune pour un grand nombre de sous-systèmes : centraliser de tels services autour d'un même et unique module pourrait être une éventuelle évolution des systèmes d'exploitation destinés au contexte *HPC*.

BIBLIOGRAPHIE

[ADR03] Dave Anderson, Jim Dykes, and Erik Riedel. More than an interface - scsi vs. ata. In *FAST*. USENIX, 2003.

[Amd67] G.M. Amdahl. Validity of the single-processor approach to achieving large scale computing capabilities. In *In AFIPS Conference Proceedings vol 30, AFIPS Press, Reston, Va,*, pages 483–485, 1967.

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

[Bac86] Maurice Bach. *The Design of the Unix 0perating System*. Prentice-Hall, 1986.

[Bak74] K.R. Baker. *Introduction to Sequencing and Scheduling*. Wiley, New York, 1974.

[Ban03] Nikhil Bansal. *Algorithms for Flow Time Scheduling*. PhD thesis, Dept. of Computer Science, Carnegie Mellon University, 2003.

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

[BCM98] Michael A. Bender, Soumen Chakrabarti, and S. Muthukrishnan. Flow and stretch metrics for scheduling continuous job streams. In *Proceedings of the 9th Annual ACM-SIAM Symposium On Discrete Algorithms (SODA'98)*, pages 270–279. Society for Industrial and Applied Mathematics, 1998.

[BGH05] Ali R. Butt, Chris Gniady, and Y. Charlie Hu. The performance impact of kernel prefetching on buffer cache replacement algorithms. In *Proceedings of the ACM International Conference on Measurement and Modeling of Computer Systems (SIGMETRICS 05)*, June 2005.

[Bou02] Christopher M. Boumenot. The Performance of a Linux NFS Implementation. Master's thesis, Worcester Polytechnic Institute, May 2002.

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

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

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

[CCC+03] A. Ching, A. Choudhary, K. Coloma, Wei keng Liao (Northwestern University), R. Ross, and W. Gropp (Argonne National Laboratory). Noncontiguous i/o accesses through mpi-io. In *Proceedings of the 3rd International Symposium on Cluster Computing and the Grid*, May 2003.

[Cec01] E. Cecchet. *Apport des réseaux à capacité d'adressage pour des grappes à mémoire partagée distribuée logicielle*. PhD thesis, Institut National Polytechnique de Grenoble, Juillet 2001.

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

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

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

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

[CGKK04] C. Chekuri, A. Goel, S. Khanna, and 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.

[CH81] Richard W. Carr and John L. Hennessy. Wsclock : a simple and effective algorithm for virtual memory management. In *SOSP '81 : Proceedings of the eighth ACM symposium on Operating systems principles*, pages 87–95, New York, NY, USA, 1981. ACM Press.

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

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

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

[CM01] F. Chen and S. Majumdar. Performance of parallel i/o scheduling strategies on a network of workstations. *Eighth International Conference on Parallel and Distributed Systems*, 2001.

[CRU03] Olivier Cozette, Cyril Randriamaro, and Gil Utard. READ 2 : Put disks at network level. In *Workshop on Parallel I/O in Cluster Computing and Computational Grids*, pages 698–704, Tokyo, May 2003. IEEE Computer Society Press. Organized at the IEEE/ACM International Symposium on Cluster Computing and the Grid 2003.

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

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

[dt] Bonnie development team. Bonnie++ benchmark suite (website). http ://www.coker.com.au/bonnie++/.

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

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

[Edi89] RFC Editor. NFS : Network File System Specification. RFC1094, March 1989.

[Edi95] RFC Editor. NFS Version 3 Protocol Specification. RFC1813, Juin 1995.

[edi03] RFC editor. Network file system (NFS) version 4 protocol, RFC 3530, April 2003.

[ES03] Daniel Ellard and Margo Seltzer. NFS tricks and benchmarking traps. In *Proceedings of the FREENIX 2003 Technical Conference*, pages 101–114, San Antonio, TX, June 2003.

[eYD05] Adrien Lebre et Yves Denneulin. aIOLi : gestion des entrées/sorties parallèles dans les grappes SMPs. In *Actes de la 16ème édition de conférence RENPAR, Le Croisic, FR*, Avril 2005.

[FK99] I. Foster and C. Kesselman. The grid : Blueprint for a new computing infrastructure. *eds. Morgan Kaufmann*, 1999.

[FRS04] Dror G. Feitelson, Larry Rudolph, and Uwe Schwiegelshohn. Parallel job scheduling — a status report. In Dror G. Feitelson, Larry Rudolph, and Uwe Schwiegelshohn, editors, *Job Scheduling Strategies for Parallel Processing*, pages 1–16. Springer Verlag, 2004. Lect. Notes Comput. Sci. vol. 3277.

[GGC+05] Luís Fabrício Góes, Pedro Guerra, Bruno Coutinho, Leonardo Rocha, Wagner Meira, Renato Ferreira, Dorgival Guedes, and Walfredo Cirne. AnthillSched : A scheduling strategy for irregular and iterative I/O-intensive parallel jobs. In Dror G. Feitelson, Eitan Frachtenberg, Larry Rudolph, and Uwe Schwiegelshohn, editors, *Job Scheduling Strategies for Parallel Processing*, pages 108–122. Springer Verlag, 2005. Lect. Notes Comput. Sci. vol. 3834.

[GH03] E. Grochowski and R. D. Halem. Technological impact of magnetic hard disk drives on storage systems, 2003.

[Gog05] Brice Goglin. *Réseaux rapides et stockage distribué dans les grappes de calculateurs : propositions pour une interaction efficace*. PhD thesis, École normale supérieure de Lyon, 46, allée d'Italie, 69364 Lyon cedex 07, France, October 2005. 194 pages.

[HBP+81] J. M. Harker, D. W. Brede, R. E. Pattison, G. R. Santana, and L. G. Taft. A quarter century of disk file innovation, 1981.

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

[HH05] Dean Hildebrand and 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.

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

[HK04] Rainer Hubovsky and 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.

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

[ID01] Sitaram Iyer and Peter Druschel. Anticipatory scheduling : A disk scheduling framework to overcome deceptive idleness in synchronous I/O. In *18th ACM Symposium on Operating Systems Principles*, October 2001.

[IMO+04] Florin Isaila, Guido Malpohl, Vlad Olaru, Gabor Szeder, and 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.

[Inc02] Cluster File System Inc. Lustre : A scalable, high performance file system, November 2002.

[IRU01] Jonathan Ilroy, Cyrille Randriamaro, and Gil Utard. Improving MPI-I/O Performance on PVFS. In *European Conference on Parallelism : EuroPar'2001*, August 2001.

[IS99] L. Iftode and J. P. Singh. Shared virtual memory : Progress and challenges. *Proc. of the IEEE, Special Issue on Distributed Shared Memory*, 87(3) :498–507, 1999.

[IT03] Florin Isaila and Walter F. Tichy. Clusterfile : a flexible physical layout parallel file system. *Concurrency and Computation*, 15(7/8) :653–679, 2003.

[JC03] David Patterson Jim Cray. A conversation with jim cray, June 2003.

[JSWB97] Ravi Jain, Kiran Somalwar, John Werth, and J. C. Browne. Heuristics for scheduling I/O operations. *IEEE Transactions on Parallel and Distributed Systems*, 8(3) :310–320, March 1997.

[Kot94] David Kotz. Disk-directed I/O for MIMD multiprocessors. In *Proceedings of the 1994 Symposium on Operating Systems Design and Implementation*, pages 61–74.

USENIX Association, November 1994. Updated as Dartmouth TR PCS-TR94-226 on November 8, 1994.

[KP97] B. Kalyanasundaram and K. Pruhs. Minimizing flow time nonclairvoyantly. In *Proceedings of the 38th Annual Symposium on Foundations of Computer Science (FOCS '97)*, 1997.

[KP00] B. Kalyanasundaram and K. Pruhs. Speed is as powerful as clairvoyant. *Journal of the ACM, Vol. 47, No. 4*, pages 617–643, July 2000.

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

[Lam78] L. Lamport. Time, clocks and ordering of events in distributed systems. *Communications of the ACM*, July 1978.

[LCC+05] W.K. Liao, K. Coloma, A. Choudhary, E. Russell L. Ward, , and S. Tideman. Collective caching : Application-aware client-side file caching. In *Proceedings of the 14th IEEE International Symposium on High Performance Distributed Computing*, July 2005.

[LD05a] Adrien Lebre and Yves Denneulin. aIOLi : gestion des entrées/sorties parallèles dans les grappes de SMPs. Technical Report Research Report RR-5522, INRIA, 38330 Montbonnot FR, Mars 2005. http ://aioli.imag.fr/DOWNLOADS/aiolimaster-report-rr5522.pdf.

[LD05b] Adrien Lebre and Yves Denneulin. aIOLi : An input/output library for cluster of SMP. In *Proceeding of the 5th International Symposium on Cluster Computing and Grid,Cardiff, UK*, May 2005.

[LDHS06] Adrien Lebre, Yves Denneulin, Guillaume Huard, and Przemyslaw Sowa. A cluster-wide adaptive i/o scheduling for concurrent parallel applications - extended abstract. In *proceedings of the 15th IEEE International Symposium on High Performance Distributed Computing, Paris, FR*, June 2006.

[LDV05] Adrien Lebre, Yves Denneulin, and Than-Trung Van. Controlling and scheduling parallel i/o in a multi-applications environment. Technical Report Research Report RR-5689, INRIA, 38330 Montbonnot FR, September 2005. http ://aioli.imag.fr/DOWNLOADS/aiolimaster-report-rr5689.pdf.

[LDVL03] Pierre Lombard, Yves Denneulin, Olivier Valentin, and Adrien Lebre. Improving the performances of a distributed nfs implementation. In *Proceeding of the 5th International Conference on Parallel Processing and Applied Mathematics, Czestochowa, Poland*, September 2003.

[Leb02] 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.

[LHS06] Adrien Lebre, Guillaume Huard, and Przemyslaw Sowa. Optimisation des E/S avec QoS dans les environnements multi-applicatif distribués. In *Actes de la 17ème édition de conférence RENPAR, Perpignan, FR, à paraître*, Octobre 2006.

[LHSD06] Adrien Lebre, Guillaume Huard, Przemyslaw Sowa, and Yves Denneulin. I/O Scheduling service for Multi-Application Clusters. In *Proceeding of the IEEE International Conference on Cluster Computing, Barcelona, SP, to appear*, Sept 2006.

[LKA04] Joseph Leung, Laurie Kelly, and James H. Anderson. *Handbook of Scheduling : Algorithms, Models, and Performance Analysis*. CRC Press, Inc., Boca Raton, FL, USA, 2004.

[LMRC04] Rob Latham, Neil Miller, Robert Ross, and Phil Carns. A Next-Generation Parallel File System for Linux Clusters. *LinuxWorld*, 2(1), January 2004.

[Lom03] Pierre Lombard. *NFSP : Une Solution de Stockage Distribué pour Architectures Grande Échelle*. PhD thesis, Institut National Polytechnique de Grenoble, Grenoble, FR, Décembre 2003.

[Lov05] Robert Love. *Linux kernel development*. Novell Press, Indianapolis, IN, USA, second edition, 2005.

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

[LS90] Eliezer Levy and Abraham Silberschatz. Distributed file systems : Concepts and examples. *Departement of Computer Sciences, University of Texas at Austin, Texas 78712-1188*, 1990.

[LSV05] Arnaud Legrand, Alan Su, and Frédéric Vivien. Minimizing the stretch when scheduling flows of biological requests. Research report RR2005-48, École Normale Supérieure de Lyon, October 2005.

[May01] John M. May. *Parallel I/O for High Performance Computing*. Morgan Kaufmann, 2001.

[MBR03] Reagan Moore, Dave Belanger, and Tom Ruwart. Petabytes and beyond (invited sessions). In *FAST '03 : Proceedings of the 2nd USENIX Conference on File and Storage Technologies*, page 44, Berkeley, CA, USA, 2003. USENIX Association.

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

[MGR03] M. Mesnier, G. Ganger, and E. Riedel. Object-based storage, 2003.

[MJoMW03] Xiasong Ma, Xiangmin Jiao, and Michael Campbell oand 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.

[MM05] M. Martinasso and J-F. Méhaut. Prediction of communication latency over complex network behaviors on SMP clusters. In *proceedings of the 2nd EPEW, vol. 3670 of LNCS, pages 172-186, 2005, Springer-Verlag*, 2005.

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

[MW00] Jeonghoon Mo and Jean Walrand. Fair end-to-end window-based congestion control. *IEEE/ACM Trans. Netw.*, 8(5) :556–567, 2000.

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

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

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

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

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

[NL01] Bill Nitzberg and Virginia Lo. Collective buffering : Improving parallel I/O performance. In Hai Jin, Toni Cortes, and 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.

[OJM06] Auréline Ortiz, Jacques Jorda, and Abdelaziz M'zoughi. Toward a new direction on data management in grids - extended abstract. In *proceedings of the 15th IEEE International Symposium on High Performance Distributed Computing, Paris, FR*, June 2006.

[PGK88] David Patterson, Garth Gibson, and Randy Katz. A case for redundant arrays of inexpensive disks (RAID). In *Proceedings of the ACM SIGMOD International Conference on Management of Data*, pages 109–116, Chicago, IL, June 1988. ACM Press.

[PST04] K. Pruhs, J. Sgall, and E. Torng. Online scheduling. In *Hanbook of Scheduling*, chapter 15. CRC Press, 2004.

[RA81] G. Ricart and A. K. Agrawala. An optimal algorithm for mutual exclusion in computer networks. *Communications of the ACM*, January 1981.

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

[Ray89] M. Raynal. A tree-based algorithm for distributed mutual exclusion. *ACM Trans. on Computer Systems (TOCS)*, 1989.

[RJTM03] B. J. Truskowski R. J. T. Morris. The evolution of storage systems, 2003.

[RKPH01] Rolf Rabenseifner, Alice E. Koniges, Jean-Pierre Prost, and Richard Hedges. The parallel effective i/o bandwidth benchmark : b_eff_io, November 2001.

[Ros00] R. Ross. Reactive scheduling for parallel i/o systems, 2000. Robert B. Ross. Reactive Scheduling for Parallel I/O Systems. PhD thesis, Electrical and Computer Engineering Dept., Clemson University.

[RS] David P. Reed and Liba Svobodova. Swallow : A distributed data storage system for a local network. In Local Networks for Computer Communication, West, A. and Janson, P., Eds., North Holland Publishing Company, 1981, pp. 355-373.

[SB02] Manish Sharma and 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.

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

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

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

[SCO90] M. Seltzer, P. Chen, and J. Ousterhout. Disk scheduling revisited. In *Proceedings of USENIX*, pages 313–323, 1990.

[SGM86] Kenneth Salem and Hector Garcia-Molina. Disk striping. In *Proceedings of the IEEE 1986 Conference on Data Engineering*, pages 336–342, 1986.

[SK85] I. Suzuki and T. Kazami. A distributed mutual exclusion algorithm. *ACM Trans. on Computer Systems (TOCS) 3*, Nov. 1985.

[SS94] M. Singhal and N. G. Shivaratri. *Advanced concepts in operating systems*. McGraw-Hill, 1994.

[SSB+95] T. Sterling, D. Savarese, D. J. Becker, J. E. Dorband, U. A. Ranawake, and C. V. Packer. BEOWULF : A parallel workstation for scientific computation. In *Proceedings of the 24th International Conference on Parallel Processing*, pages I :11–14, Oconomowoc, WI, 1995.

[Sto98] Heinz Stockinger. Dictionary on parallel input/output. Master's thesis, Department of Data Engineering, University of Vienna, February 1998.

[Tan01] A. S. Tanenbaum. *Modern operating systems*. Prentice Hall, 2001.

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

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

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

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

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

[Tho04] Alexander Thomasian. *Handbook of Scheduling : Algorithms, Models, and Performance Analysis*. CRC Press, Inc., Boca Raton, FL, USA, 2004.

[TLG97] Rajeev Thakur, Ewing Lusk, and William Gropp. Users guide for ROMIO : A high-performance, portable MPI-IO implementation, revised may 2002. Technical Report ANL/MCS-TM-234, Mathematics and Computer Science Division, Argonne National Laboratory, October 1997.

[Van97] Rodney Van Meter. Observing the effects of multi-zone disks. In *Annual USENIX Technical Conference (Anaheim, CA)*, pages 19–30, January 1997.

[Van05] Than-Trung Van. Service de régulation et d'ordonnancement de requêtes d'e/s au sein des architectures parallèles. Master's thesis, Joseph Fourier University at Grenoble (France), June 2005.

[VSK+03] Murali Vilayannur, Anand Sivasubramaniam, Mahmut Kandemir, Rajeev Thakur, and Robert Ross. Discretionary caching for i/o on clusters. In *IEE, 3st International Symposium on Cluster Computing and the Grid*, May 2003.

[WNH01] B. White, W. Ng, and B. Hillyer. Performance comparison of IDE and SCSI disks, 2001.

[Wol98] R. Wolski. Dynamically forecasting network performance using the network weather service. *appeared in Cluster Computing : Networks, Software Tools, and Applications*, Jan. 1998.

[ZYSM03] Yanyong Zhang, Antony Yang, Anand Sivasubramaniam, and Jose Moreira. Gang scheduling extensions for I/O intensive workloads. In Dror G. Feitelson, Larry Rudolph, and Uwe Schwiegelshohn, editors, *Job Scheduling Strategies for Parallel Processing*, pages 183–207. Springer Verlag, 2003. Lect. Notes Comput. Sci. vol. 2862.

CALCUL DU DEGRÉ DE CONTINUITÉ

A

Le temps nécessaire pour accéder aux informations varie selon les méthodes d'optimisation mises en place. L'utilisation de techniques visant à agréger les accès améliore nettement les performances. Toutefois dans certains cas critiques, l'approche habituelle UNIX s'avère moins coûteuse.

[Raj02] a défini une valeur, appelée «degré de continuité», qui permet de quantifier si une stratégie de type agrégation doit être ou pas mise en œuvre.

Cette annexe décrit comment cette valeur est calculée.

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

Quantification au sein d'un nœud :

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

$$T_1 = n\left(S + \frac{C}{D}\right) \quad \begin{array}{l}\text{avec } S\text{, temps de positionnement sur l'}\textit{offset}\ (\texttt{seek}),\\ C \text{ taille d'un bloc en Mo,}\\ D \text{ débit en Mo/s et } n \text{ nombre total de blocs.}\end{array} \tag{A.1}$$

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

$$T'_1 = S + \frac{C'}{D} + nM \quad \begin{array}{l}\text{avec } C' \text{ taille du bloc de recouvrement en Mo}\\ \text{(au maximum la taille du tampon temporaire)}\\ \text{et } M = \frac{C}{Q} \text{ temps nécessaire pour la copie temporaire d'un bloc}\\ \text{utile (}\texttt{memcpy}\text{).}\end{array} \tag{A.2}$$

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

$$T'_1 \ll T_1 \quad \equiv \quad S + \frac{C'}{D} + n\frac{C}{Q} \ll n\left(S + \frac{C}{D}\right) \tag{A.3}$$

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

$$\frac{C'}{D} \gg n\left(S + \frac{C}{D}\right) \quad \Rightarrow \quad T'_1 \gg T_1 \tag{A.4}$$

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

$$T'_1 = n\left(S + \frac{C'}{D} + \frac{C}{Q}\right) \quad avec \quad C' \geq C \quad \Rightarrow \quad T'_1 \gg T_1 \tag{A.5}$$

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

Quantification au sein d'un groupe de processus :

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

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

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

(A.6)

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

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

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

(A.7)

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

$$T'_n = pn\frac{R}{B} + pT'_1 + pn\frac{L}{B} \equiv T'_n = n\frac{R}{B} + pT'_1 + n\frac{L}{B} \tag{A.8}$$

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

$$T'_n = n\frac{R}{B} + T'_1 + n\frac{L}{B} \tag{A.9}$$

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

Mise en place d'un coefficient :

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

$$\beta = 1 - \frac{min(T'_1, T'_n)}{T_S} \tag{A.10}$$

De cette manière :

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

A ce jour, cette estimation n'est incluse que dans la bibliothèque *Mercutio* (cf. section A.2) La bibliothèque `ROMIO` se base sur un calcul élémentaire (cf. section A.2).

Résumé
aIOLi : Contrôle, Ordonnancement et Régulation des Accès aux Données Persistantes dans les Environnement Multi-applicatifs Haute Performance

De nombreuses applications scientifiques utilisent et génèrent d'énormes quantités de données. Ces applications qui exploitent des modèles d'accès parallèles spécifiques (principalement des accès disjoints) sont souvent pénalisées par des systèmes de stockage inadaptés. Pour éviter les dégradations de performances, les bibliothèques d'Entrées/Sorties parallèles telles que *ROMIO* sont généralement utilisées pour agréger les petites requêtes séparées en de plus grosses contiguës habituellement plus performantes. Toutefois, les optimisations apportées pour un programme ne tiennent pas compte de l'ensemble des interactions avec d'autres applications s'exécutant en concurrence sur la grappe. La conséquence est que ces routines spécifiques visant à optimiser les accès d'une application vont s'avérer inutiles, car leur effet va être perturbé par les autres applications !

Ce document décrit une nouvelle approche, appelée *aIOLi*, permettant le contrôle, le réordonnancement et la régulation de l'ensemble des interactions générées par les différentes applications s'exécutant simultanément sur une grappe et ce, en s'appuyant uniquement sur l'interface `POSIX`.

Dans un tel contexte, la performance, l'interactivité et l'équité sont des critères pour lesquels il est important de trouver un bon compromis. Pour y parvenir, une stratégie d'ordonnancement globale prenant en compte également les problématiques d'Entrées/Sorties parallèles locales aux applications a été définie. Le service *aIOLi* consiste en un support d'ordonnancement générique pouvant être rattaché à différentes parties d'un système de fichiers. L'exécution concurrente de jeux de tests *IOR* sur un serveur NFS traditionnel ont montré des améliorations particulièrement significatives pour les accès en lecture en comparaison aux performances pouvant être atteintes avec les routines `POSIX` ou `MPI I/O`.

Mots-clés : E/S parallèle, calcul intensif, `MPI I/O`, grappe.

Abstract
aIOLi : I/O Scheduling Service for Multi-Applications Clusters

Lots of scientific applications use and create vast amounts of data. Those often have specific ways to access data in non-sequential patterns (strided requests). To avoid performance loss, parallel I/O libraries such as *ROMIO* are often used to aggregate small separate requests into large contiguous ones. However, optimizations for a given applications are not aware of the whole set of interactions with other ones running at the same time on the cluster. As a consequence, most of the optimization work is lost because they will be disturbed by the other applications !

This document presents a software service, named *aIOLi*, whose role is to control, reschedule and regulate the whole set of interactions coming from all applications running simultaneously on a cluster. Besides, the traditional `POSIX` API is maintained and used.

In such a context, trade-off have to be found between performance, fairness and response time. To achieve this, an I/O scheduling algorithm together with a "*requests aggregator*" considering both application access patterns and global system load have been designed and merged into *aIOLi*. The *aIOLi* service consists of a new generic framework pluggable into any I/O file system. Several concurrent runs of the *IOR* benchmarks show significant improvements on read accesses with regards to `POSIX` and *ROMIO* calls.

Keywords : Parallel I/O, High Performance Computing, `MPI I/O`, cluster.