INSTITUT NATIONAL DE RECHERCHE EN INFORMATIQUE ET EN AUTOMATIQUE

aIOLi : gestion des Entrées/Sorties Parallèles dans les grappes SMP

Adrien Lebre — Yves Denneulin

N° 5522

Mars 2005

Thème NUM

ISSN 0249-6399

aIOLi : gestion des Entrées/Sorties Parallèles dans les grappes SMP*

Adrien Lebre , Yves Denneulin

Thème NUM — Systèmes numériques
Projet MESCAL

Rapport de recherche n° 5522 — Mars 2005 — 12 pages

Résumé : Un grand nombre d'applications scientifiques (biologie, climatologie, ...) utilisent et génèrent des quantités de données qui ne cessent de croître en plus de modes d'accès parallèles qui leur sont particuliers. Aux problématiques "out-of-core" ou "parallel I/O" longuement abordées dans un contexte d'accès local, viennent s'ajouter les considérations et les contraintes imposées par un environnement distribué comme l'est une grappe. Plusieurs approches ont été proposées à la communauté scientifique dans le but d'améliorer les performances lors d'accès parallèles à des fichiers distants (systèmes de fichiers ou encore librairies d'entrées/sorties spécialisées). Toutefois, ces solutions intègrent des API plus ou moins lourdes mais surtout spécifiques qui nécessitent une connaissance exacte de chaque subtilité interne au modèle. Ce papier présente la solution "aIOLi" : une librairie transparente et efficace pour les accès parallèles aux fichiers au sein d'une grappe SMP. En intervenant au niveau des points de concentration, aIOLI exploite des algorithmes d'Entrées/Sorties parallèles sans pour autant faire appel à des mécanismes de synchronisations inter-processus et en s'appuyant simplement sur les interfaces standard de la librairie C (`open/lseek/read/write/close`). Les performances obtenues avec notre prototype atteignent les limites du système de stockage distant.

Mots-clés : aIOLi, Entrées/Sorties Parallèles, HPC, grappe, SMP, système de fichiers

* Ce travail a été effectué dans le cadre du projet MESCAL (CNRS, INPG, INRIA et UJF) et du projet LIPS (BULL et INRIA). La grappe IDPOT (`http://idpot.imag.fr`) a été utilisée pour les expérimentations.

Unité de recherche INRIA Rhône-Alpes
655, avenue de l'Europe, 38334 Montbonnot Saint Ismier (France)
Téléphone : +33 4 76 61 52 00 — Télécopie +33 4 76 61 52 52

aIOLi : An Input/Output LIbrary for cluster of SMP

Abstract: As clusters use grows, lots of scientific applications (biology, climatology, nuclear physics ...) have been rewritten to fully exploit this extra CPU power and storage capacity. This kind of software uses and creates huge amounts of data with typical parallel I/O access patterns. Several issues, like "out-of-core limitation" or "efficient parallel input/output access" already known in a local context (on SMP nodes for example), have to be handled in a distributed environment such as a cluster. The effective local hardware facilities which reduced response time and access constraints on SMP could not provide optimal performances with respect to CPU and network power available in a cluster. Several solutions have been proposed by the scientific community to handle these issues, like *Parallel File systems* or *Parallel I/O Libraries*, but their specific API limits portability and requires good knowledge of their internal mechanisms. This paper presents an efficient I/O library, aIOLi, for parallel access to remote storages in SMP clusters. Thanks to the SMP kernel features, our framework provides parallel I/O without inter-processes synchronisation mechanisms as well as a simple interface based on the classic UNIX system calls (`creat/open/read/write/close`). Our experiments show that the aIOLi solution allows us to achieve performance close to the limits of the remote storage system.

Key-words: aIOLi, Parallel I/O, cluster, SMP, HPC, File Systems

Table des matières

1 Introduction

La gestion efficace des Entrées/Sorties est un problème récurrent au sein des architectures informatiques. Dès 1967, [4] imputait les limitations de performance vis à vis du système de stockage. Presque 40 ans plus tard, [13] confirme la disparité entre les performances des périphériques de stockage et la puissance CPU disponible au sein des nouvelles architectures. Cet écart est par ailleurs amplifié par les applications scientifiques modernes (biologie moléculaire, climatologie, physique des hautes énergies, …) qui exploitent et génèrent des quantités croissantes de données. De plus, les modes d’accès parallèles mis en œuvre dans ce type d’application affectent considérablement l’efficacité des systèmes de stockages sous-jacents.

Afin d’illustrer ces propos, nous allons analyser une opération commune consistant à récupérer des données lors d’une multiplication parallélisée de matrices : chaque processus doit récupérer des parties spécifiques de la ressource distante en fonction du découpage prédéfini(colonne, ligne, BLOCK/BLOCK, BLOCK/CYCLIC, …). La première approche se décompose 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” nécessite d’une part un réseau de communication performant en vue de de limiter les surcoûts imposés par le transfert redondant des données et d’autre part une quantité de mémoire significative afin de palier aux problèmes de type “out-of-core”. Une seconde méthode consiste à récupérer directement et à partir de chaque processus les données désirées. Toutefois, le serveur distant reçoit en parallèle plusieurs requêtes à différents offsets et de différentes tailles ; ce type de comportement affecte considérablement les performances[1] en plus de générer des phénomènes de congestion au niveau du système de fichiers. Certes, les dernières technologies comme le RAID ou les NAS réduisent ces problèmes en exploitant des techniques d’équilibrage de requêtes. Cependant elles restent des solutions onéreuses en plus de nécessiter des intergiciels complexes permettant d’exploiter totalement leur potentiel. En outre, la répartition des données[2] sur plusieurs serveurs de stockages complexifie la correspondance entre vue logique et vue physique, [22]. Les applications utilisant des schémas d’accès récurrents de manière à optimiser les performances (comme pour la multiplication d’une matrice par exemple) peuvent se révéler nettement moins efficace et engendrer de nombreux problèmes (accès concurrents, cohérence, “faux-partage”, …).

[1] L’efficacité est obtenue lors de large accès contigu sur la plupart des périphériques de stockage.
[2] “data striping”.

Un grand nombre de solutions prenant en compte les considérations précédemment énoncées (matérielles et logicielles) a été soumis à la communauté scientifique. Toutefois, les systèmes présentés proposent une multitude de fonctionnalités qui complique aussi bien les phases de développement que la maintenance de certaines parties des applications scientifiques. De plus, ils nécessitent de profondes connaissances de toutes les subtilités de leur API afin d'en tirer un gain maximal. De notre point de vue, les caractéristiques suivantes doivent être prises en compte : indépendance vis à vis des langages de programmation parallèle (comme MPI, HPF ou encore C++), indépendance vis à vis des systèmes de stockages, efficacité des accès en intégrant les algorithmes basiques d'optimisations d'Entrées/Sorties Parallèles, minimiser l'intrusion vis à vis de l'application et vis à vis de la plate-forme d'exécution. A ce jour et à notre connaissance, aucun système ne répond à ces objectifs. C'est ce constat qui a motivé le développement de la libraire aIOLi que nous présentons dans ce papier.

aIOLi est une librairie d'Entrées/Sorties disques incluant plusieurs concepts émanant des algorithmes d'Entrées/Sorties parallèles sans pour autant requérir à une API spécifique. Par ailleurs, aIOLi élimine l'ensemble des synchronisations nécessaires et coûteuses dans les algorithmes collectifs en intervenant au niveau des points de concentration. La suite du document est organisée de la manière suivante : un résumé succinct des solutions disponibles et de leurs limitations figure au sein de la section 2. La section 3 présente globalement notre système ; les aspects plus techniques sont abordés en section 4. Les premiers résultats obtenus avec notre prototype sont présentés par le suite, 5. Avant de conclure, nous présentons, au sein de la section 6, diverses approches basées sur des concepts plus ou moins différents mais ayant toujours pour but l'amélioration des performances des Entrées/Sorties. La partie 7 conclut ce papier et dévoile plusieurs pistes pour nos travaux à venir.

2 Contexte

Les applications exploitant de manière intensive les systèmes de fichiers distants ont imposé plusieurs contraintes dues aux architectures de type grappe d'une part mais également à cause de leur mode d'accès particulier. Plusieurs travaux préliminaires portant sur l'analyse des comportements des Entrées/Sorties au sein de ces logiciels, [11, 21], ont permis de découvrir des schémas d'accès récurrents : une multitude d'accès de faible taille et disjoints. Ce type de requête diminue considérablement les performances du système lorsqu'elles sont en grand nombre et de plus, en parallèle. Deux catégories permettent de définir les solutions tentant de réduire les phénomènes de congestion et d'améliorer les performances : les systèmes de fichiers dits "Parallèles" et les librairies spécialisées dans ce type d'accès. La première solution consiste à proposer un système complet capable de gérer les requêtes depuis l'application cliente aux zones de stockage physique. La seconde possibilité se concentre sur l'élaboration de techniques visant à réduire le nombre d'accès non contigus sans pour autant altérer la portabilité des applications.

2.1 Systèmes de fichiers "Parallèles"

Les approches centralisées comme le populaire NFS, ne répondant pas aux exigences imposées par les nouvelles applications dans la cadre des grappes, plusieurs systèmes "plus distribués" ont été conçus. PVFS [16], NFSp [17] ou encore GPFS [24] sont des solutions globales répertoriées comme systèmes de fichiers "parallèles". En s'appuyant sur une répartition des données sur plusieurs espaces de stockage distincts, ils réalisent un équilibrage de l'ensemble des requêtes et réduisent ainsi les problèmes de congestion. Certains systèmes plus spécifiques aux Entrées/ Sorties Parallèles [9, 19, 20], proposent en plus de ces techniques de répartitions, des routines exploitant aussi bien les vues logiques que le placement physique des données. Cette gestion plus fine des accès offre de réelles performances. Toutefois, l'implantation d'applications scientifiques avec de telles approches peut s'avérer fastidieuse et nécessite parfois l'écriture de code dédié. De plus, les contraintes générées par les grappes (hétérogénéité, tolérance aux pannes, spécificités matérielles, sécurité, passage à l'échelle, ...) conduisent à des modèles de plus en plus complexes et difficiles à administrer. Le dernier projet tentant de répondre à toutes ces exigences et le système de fichiers Lustre, [25]. Il s'attarde aussi bien à la problématique de l'efficacité dans les accès parallèles[3] qu'aux considérations plus génériques des systèmes de fichiers basés sur des architectures matérielles performantes (technologies RAID, réseau efficace comme quadrics, myrinet,...). Malheureusement, ce système est en cours de développement et un grand nombre de fonctionnalités n'est pas encore opérationnelle.

2.2 Librairies d'Entrées/Sorties Parallèles

Afin de réduire les dépendances liées au système de stockage sous-jacent et augmenter le facteur de portabilité des applications, plusieurs librairies [7, 26, 14, 5, 18, 27], ont été développées. Moins intrusive et ne nécessitant pas d'administration particulière, cette méthode consiste à mettre en œuvre des algorithmes efficaces (agrégation/recouvrement/ré-

[3] Un module connectant le client Lustre avec ROMIO est fourni.

ordonnancement des requêtes) au dessus des couches clientes des systèmes de fichiers qui gardent la responsabilité de la gestion matérielle[4]. En 1997, dans un souci d'uniformisation des interfaces, le consortium MPI a défini les spécifications MPI I/O [8, 1]. Ce standard décrit en détails plusieurs routines permettant de mieux appréhender les Entrées/Sorties parallèles au sein des programmes MPI. Plusieurs implantations du modèle ont été réalisées, les plus connues sont ROMIO [27, 2] et, plus récemment, MercutIO[5].[23]. En plus de routines spécifiques permettant de récupérer des informations particulières vis à vis du système de stockage sous-jacent, ces implantations exploitent deux techniques majeures améliorant les performances dans le contexte des Entrées/Sorties Parallèles : le "Data Sieving" et l'approche "Two Phases" [27]. L'algorithme "Data Sieving" réduit le nombre de requêtes transmises par un client. L'idée consiste à exploiter un masque de fichier défini au sein du code de l'application afin de traiter plusieurs accès disjoints par une unique demande recouvrante. L'approche "Two-Phases" est une technique collective[6]. qui consiste à échanger des informations préalablement avec l'ensemble des clients afin de déterminer la meilleure solution d'accès. Durant la première phase, une répartition efficace des accès est déterminée ; il en résulte une requête recouvrante pour chaque processus. La seconde étape consiste à redistribuer aux clients adéquats, les données souhaitées. Dépendant de la taille d'accès, ce genre d'approche ne requiert pas obligatoirement de masque de fichiers puisque à chaque appel collectif, les informations concernant l'ensemble des requêtes sont échangées. Toutefois, comme [3] le précise, ces approches collectives engendrent des mécanismes de synchronisations onéreux. Ces solutions proposent différentes techniques permettant d'améliorer dans des cas précis la gestion des Entrées/Sorties Parallèles. Néanmoins, elles nécessitent une compréhension rigoureuse de chacune des subtilités qu'elles proposent (création de types structurés, définition de masque d'accès, ...) en plus d'augmenter considérablement le temps de développement. Les fonctionnalités basiques du standard MPI I/O (`MPI_File_read()` ou `MPI_File_write()`) ne fournissent aucune amélioration par rapport aux appels `read()` ou `write()` et au contraire engendrent des surcoûts dues aux copies intempestives entre le client, la librairie et le noyau.

3 Présentation du modèle

Depuis plusieurs années, notre équipe mentionne l'importance de prendre en considération les modèles et les interfaces largement établies dans la conception de nouvelles solutions. Une version parallélisée d'un serveur NFS exploitable à partir de la couche cliente standard, [17], a par exemple été proposée. Poursuivant nos travaux dans cette même direction, nous voulons offrir une solution efficace pour la gestion des Entrées/Sorties Parallèles en s'appuyant uniquement sur les appels standards de la bibliothèque C. Cette approche permet d'assurer la portabilité de notre système sur la quasi totalité des architectures informatiques.

3.1 Étude préliminaire

Une caractéristique fondamentale des architectures SMPs est que quelque soit le nombre de processeurs qui la compose, un unique noyau contrôle l'ensemble des évènements liés aux "Entrées/Sorties fichiers". Dans l'approche classique du traitement d'une requête, le client "transmet" sa demande à la pile d'Entrées/Sorties du noyau qui l'exécute et renvoie le résultat au processus concerné. Dans les architectures parallèles, il est possible d'avoir plusieurs requêtes émanant de différents processus clients au sein de la pile d'Entrées/Sorties. Malheureusement, ces demandes peuvent être traitées parallèlement[7] sans analyser si des techniques de type "Data Sieving" ou "Two Phases" seraient susceptibles de réduire le temps de complétion global. Au contraire, la gestion parallèle des diverses requêtes va dégrader les performances en plus de saturer rapidement le noyau. Le graphique 1 illustre ce type de comportement : le temps nécessaire pour récupérer un fichier distant de 1Go à partir d'un nombre variable de processus au sein d'un même nœud SMP a été mesuré. Cette expérimentation a été exécutée sur la grappe idpot disponible au laboratoire ID (48 machines Bi-Xeon 2.5 GHz avec chacune 1.5 Go de RAM et interconnectées par un réseau gigabit ethernet). L'évaluation a été effectuée sur 1, 2, 4 et 8 instances d'un programme MPI permettant de mettre en œuvre une décomposition du fichier[8]. La granularité des accès varie de de 1Ko à 4Mo[9].

Même si une corrélation semble apparaître entre la granularité et le nombre de processus concernés, nous voulons juste faire remarquer comment les performances sont intimement liées au nombre de clients participant à la décomposition. Par ailleurs, la dernière courbe révèle que pour des accès supérieure à 128Ko, l'envoi séquentielle des requêtes, et ce même dans le désordre, est plus efficace que le traitement en parallèle des requêtes. Ce constat peut s'expliquer par le fait d'un

[4] Les aspects fondamentaux devant être pris en considération pour la conception de systèmes de fichiers "modernes" sont mentionnés dans [28].

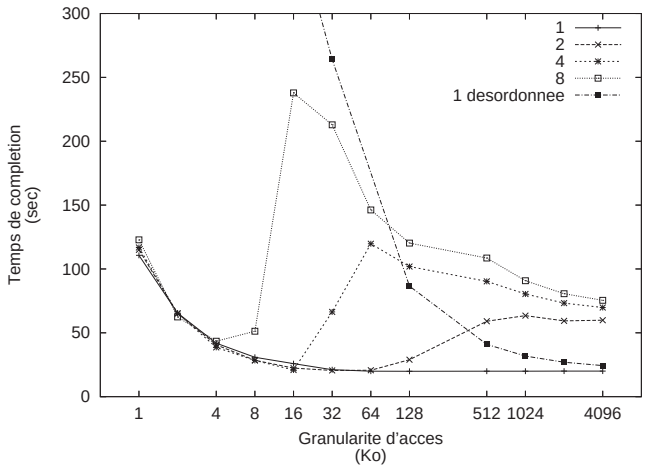
[5] MercutIO a été incorporée à une version commerciale de MPI intitulée ChaMPIon.

[6] Plusieurs approches similaires ont été proposées : [15, 31][26].

[7] Plusieurs parties du noyau pouvant s'exécuter au même instant.

[8] Le test a été implanté sous un noyau Linux 2.4.27 avec la bibliothèque *mpich 1.2.5* et les appels traditionnels POSIX. Le serveur distant est un système NFS version 3 dédié à notre expérimentation (`nfsvers=3,tcp,rsize=32768,noac`).

[9] La borne supérieure de 4 Mo a été choisie en fonction des paramètres de la bibliothèque ROMIO, cf. 5.2.



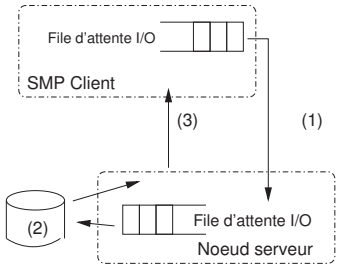
Les quatre premières courbes correspondent aux différentes exécutions avec 1, 2, 4 et 8 instances MPI. La dernière courbe représente la lecture du même fichier par un seul client mais dans un ordre aléatoire ; alors que la première consiste en fait en une lecture séquentielle du fichier.

FIG. 1 – Décomposition d'un fichier de 1Go

mauvais ordonnancement des demandes du côté du système de fichiers : lorsque le noyau délivre 8 requêtes en parallèle, le meilleur ordonnancement requiert 1 déplacement (afin de placer le pointeur de fichier à l'offset de démarrage) suivi de 8 lectures consécutives. Dans le pire des cas, l'ordonnancement est susceptible d'engendrer $nb\_requete + 1$ déplacements en plus des $nb\_requete$ lectures. Ce comportement peut être amplifié selon les mécanismes internes exploités par le système de fichiers. Dans notre cas ; le client NFS découpe chaque requête en sous accès de 32Ko augmentant ainsi le nombre potentiel de déplacements sur le serveur. Par conséquent, transmettre les requêtes une à une, tend à réduire ce phénomène (chaque sous requête étant traitée de manière séquentielle et contiguë).

3.2 Principes de la solution aIOLi

A partir du bilan établi précédemment, une solution naïve permettant d'améliorer les performances consisterait à stocker temporairement les requêtes afin de les délivrer en tenant compte des contraintes de temps et de la charge du serveur distant, figure 2.



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

Afin de maximiser l'utilisation du sous système de stockage, il est nécessaire que la prochaine demande soit présente au sein de la file d'attente du serveur au plus tard à la fin de l'étape (2). Ainsi grâce à la taille de la dernière demande délivrée et la charge du serveur, il est possible de déterminer une fenêtre de tir pendant laquelle un ordonnancement plus efficace peut être analysé du coté client ; le tout étant de transmettre dans les temps, la prochaine demande.

FIG. 2 – Les accès vers un système de stockage distant

Notre solution consiste à mettre en œuvre pendant la période qui sépare deux envois, des optimisations de type recouvrement/agrégation/ré-ordornancement permettant d'améliorer les performances. Par exemple, une technique dérivée d'agrégation des accès (comme "Data Sieving" ou "Two Phases") peut facilement être implantée au niveau d'un point de concentration comme l'est la pile des entrées/sorties d'un système d'exploitation et ce sans nécessiter la définition de schéma d'accès, de points de synchronisations ou encore d'échanges d'informations. Entre chaque envoi, la file d'attente coté client est analysée dans le but de découvrir des accès séquentiels ou même contigus. Les nouvelles requêtes ainsi déterminées sont alors ordonnancées par rapport à l'offset de départ et les contraintes de temps (maintien de la cohérence). Cependant comme nous l'avons brièvement mentionné durant l'introduction, ces techniques sont dépendantes du placement physique des données. La forte introduction dans les grappes de systèmes de fichiers parallèles comme PVFS et Lustre peu à peu, engendrent de nouvelles difficultés dues aux méthodes de répartition des données. Ces dernières peuvent rendre inutiles voir même coûteuses les approches de type recouvrement/agrégation. Il est donc important d'intégrer ces contraintes à notre modèle. Malheureusement, nous recherchons des méthodes n'altérant pas la transparence ou la portabilité de notre modèle. En effet, les fonctions de type "hint" employées dans les implantations de MPI I/O ne répondent pas à ces exigences. La proposition actuelle consiste à configurer le service aIOLi avec le facteur de répartition et le nombre

de serveurs de stockage. Cette première solution permet d'éliminer les agrégations/recouvrements inutiles. Toutefois, elle ne permet pas de déterminer le serveur en charge de la requête et, par conséquent, rend impossible une transmission tenant compte du temps ainsi que de la charge courante. Deux paramètres supplémentaires sont nécessaires pour y parvenir : le premier serveur de stockage utilisé et la politique de répartition exploitée. Si le second paramètre généralement statique peut être renseigné lors de la configuration du système, le premier dépend souvent de la fonction de hachage intrinsèque au système de fichiers. Certes, les récents systèmes proposent des routines résolvant cette correspondance mais nous ne souhaitons pas les employer puisque cela conduirait à implanter une couche d'abstraction pour les divers systèmes, ce qui, une fois de plus, ne répond pas à nos critères.

4 Notes techniques et implantation

Cette section présente les aspects techniques du prototype actuel. Afin de simplifier l'implantation (le but étant de valider notre approche), le modèle a été développé en espace utilisateur.

4.1 Architecture

L'architecture globale est présentée au sein de la figure 3 : un module surchargeant les appels standards[10], `open` / `creat` / `read` / `write` / `lseek` / `close`, est lié à l'application de manière à les rediriger vers le démon aIOLi, composant principal de notre architecture. Les requêtes "postées" par les clients au sein d'une file de messages IPC globale (1) sont récupérées puis triées une premier fois par le thread de réception. Un ensemble de thread, "les threads I/O", analyse les différentes files de requêtes (2) et réalise les appels systèmes conséquents (3).

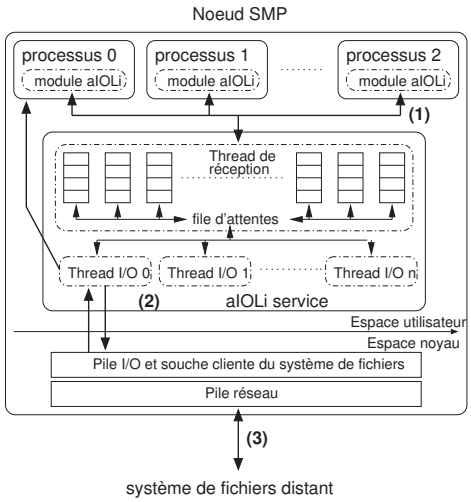


FIG. 3 – Architecture du prototype aIOLi

Il y a actuellement 6 types de requêtes classés en 2 groupes : le premier comprend les demandes qui peuvent être traitées de manière indépendante (INIT_REQ, OPEN_REQ, CLOSE_REQ, FINALIZE_REQ). Chaque requête est stockée dans une file d'attente correspondant à son type. Les "threads I/O" traitent ces demandes selon un ordre défini : tout d'abord, les requêtes d'initialisation (INIT_REQ) suivies par les ouvertures de fichiers (OPEN_REQ), puis les fermetures (CLOSE_REQ). Enfin, les requêtes de détachement (FINALIZE_REQ) sont exécutées. Cette hiérarchie est un moyen simple pour augmenter potentiellement les performances : l'initialisation entraîne forcement une ouverture qui à son tour est susceptible de générer des accès en lecture (READ_REQ) et/ou écriture (WRITE_REQ) augmentant ainsi les chances d'agrégation/recouvrement. Ce sont ces deux derniers types de requêtes (READ_REQ et WRITE_REQ) qui composent la seconde classe : pour chaque fichier ouvert, une structure spécifique est attribuée afin d'y stocker les requêtes d'accès. Les files d'attente utilisées sont triées selon l'offset de positionnement, la taille et l'estampille de temps nécessaire pour maintenir la cohérence.

La mise en œuvre du service aIOLi est réalisée lors de la première demande d'ouverture/création déposée par un client. Ce dernier va essayer de déposer une requête d'initialisation afin de souscrire au service ; ne trouvant pas la file de messages globale, il va demander au système de lancer le démon. Lors de l'initialisation, aIOLi créé l'ensemble des

[10] Les autres appels comme `dup/readv/writev/...` seront disponibles dans une version ultérieure.

structures nécessaires à la gestion des requêtes entrantes ainsi qu'au stockage des informations concernant les processus clients. Un tampon global situé en mémoire partagée est utilisé afin de réduire les copies intempestives entre les clients, aIOLi et le noyau lors des opérations de lectures ou d'écritures. Une fois la file de messages globale activée, le client a la possibilité de transmettre une requête d'initialisation contenant l'identifiant du processus ainsi qu'une clé particulière. Ces informations vont permettre au démon de mettre en place une file de messages spécifique depuis le service et à destination du client (les réponses sont transmises par ce canal au client). Le service aIOLi délivre alors un acquittement au client. L'architecture est alors en place et le processus peut utiliser les appels standards en connexion avec le service aIOLi durant toute son exécution. Lors de la terminaison du client, une requête FINALIZE_REQ est postée afin de retirer les références à des fichiers encore ouverts et de supprimer la totalité des informations le concernant au sein du démon. La réponse est transmise au client et la file de messages spécifique est supprimée.

4.2 Appels `open / creat / close`

Les appels,`open()` et `creat()`, sont simplement déposés au sein de la file de messages global pour le démon aIOLi. A la différence du standard MPI I/ où toutes les ouvertures sont collectives, aucun mécanisme de synchronisation n'est requis. Le thread de réception place la requête dans la file d'attente OPEN_REQ. Le prochain thread I/O inactif va traiter la demande d'ouverture : si le fichier a été préalablement ouvert[11] par un autre processus, le démon retourne directement le descripteur de fichier au processus client. Dans le cas contraire, le service exécute le "syscall" `open/creat` correspondant : si l'appel est valide, le descripteur est stocké dans les structures d'aIOLi et un lien depuis le PID du client vers le descripteur est enregistré. Dans tous les cas le résultat de l'appel est retourné au client. Un tableau, alloué au sein de la souche cliente du système aIOLi, enregistre la correspondance entre les descripteurs retournés par le démon et les descripteurs locaux[12].

La demande de fermeture est également postée au sein de la file de messages globale. A la réception de telle requête, le démon aIOLi retire le lien, présent au sein de ces structures, entre le PID et le descripteur de fichier. Si le descripteur n'est plus référencé, aIOLi ferme réellement le fichier.

4.3 Appel `lseek`

L'appel `lseek()` est traité de manière locale. A moins que les arguments passés lors de l'appel soient erronés, la souche cliente aIOLi met à jour le descripteur de fichier à la position désirée au sein du tableau correspondant (aucun "syscall" n'est généré). Cette valeur sera exploitée lors des prochaines opérations de lectures ou d'écritures.

4.4 Appel `read`

Lorsqu'un accès en lecture est posté, le thread de réception le place dans la file d'attente correspondante en tenant compte de l'offset et de la taille. Pour éviter les problèmes de cohérence, une liste chaînée est utilisée afin de maintenir un ordre chronologique entre les lectures et les écritures. Le prochain thread I/O inactif, après avoir vérifier qu'aucune requête prioritaire n'est présente, va analyser la file d'attente de lecture. Si celle ci contient plus d'une requête, la tâche va essayer de déterminer la meilleure politique de gestion (toujours en tenant compte des offsets, des tailles et des estampilles de temps). Tout d'abord, le thread vérifie si une autre demande débute à un offset inférieur et si cette requête a bien été estampillée avec un temps antérieur à celui de la prochaine écriture. Si ces deux conditions sont vérifiées, la requête est alors sélectionnée. La seconde étape consiste à analyser si d'éventuelles demandes peuvent être agrégées avec la présente requête [13]. Le résultat de cette étape est un vecteur d'accès (comparable à ceux définis dans l'appel `readv()`). Un unique appel recouvrant est exécuté. Afin d'éviter les copies intempestives, le tampon utilisé lors de l'appel `read()` correspond à une partie définie de la mémoire partagée. Le résultat de l'appel ainsi que l'adresse mémoire, nécessaire à la récupération des données souhaitées, sont retournés à chacun des clients concernés par la requête recouvrante.

4.5 Appel `write`

L'implantation de l'appel `write()` est en cours de finalisation. La méthode est similaire à un accès de type lecture : le client réalise une copie des données au sein d'une zone déterminée de la mémoire partagée. Il poste alors une demande d'écriture en précisant la taille et l'adresse mémoire permettant d'accéder aux données. Comme pour une lecture, la demande est placée dans la file d'attente correspondante selon les mêmes critères de tri. Un thread I/O parcours la queue

[11] Une comparaison sur le chemin complet et sur les "flags" ouverture/création est réalisée.
[12] Les descripteurs de fichiers peuvent différer. De plus, l'appel `dup()` nécessitera ces informations.
[13] Les files étant triées, il suffit de les parcourir, en arrière puis en avant, pour connaître les demandes contiguës.

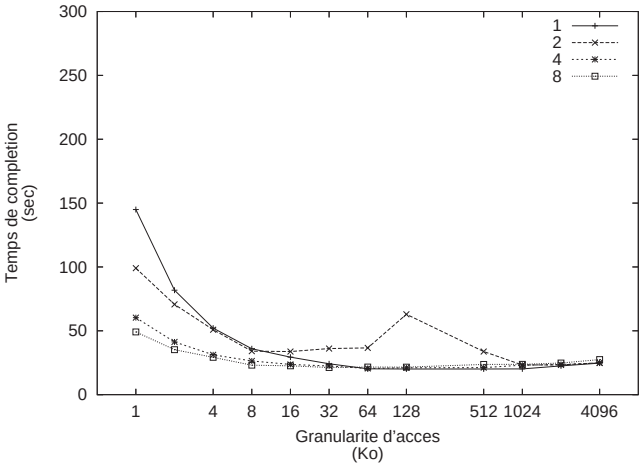


FIG. 4 – Décomposition avec aIOLi

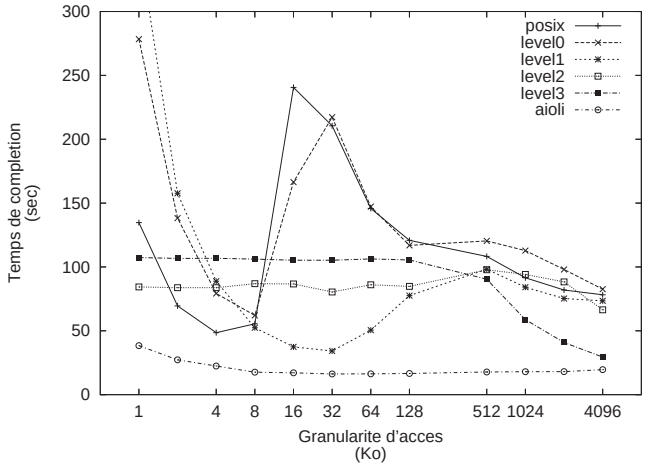


FIG. 5 – ROMIO 4 levels vs aIOLi

afin de découvrir si certaines requêtes sont contiguës et peuvent être par conséquent agrégées[14]. Si tel est le cas, une copie supplémentaire est effectuée afin de récupérer les données au sein d'un unique tampon nécessaire à l'appel `write()`. Dans le cas contraire, le tampon situé en mémoire partagée est directement utilisé pour l'opération d'écriture. Le résultat retourné par la fonction est transmis à l'ensemble des processus participant. Plusieurs optimisations comme les techniques d'écritures retardées sont envisageables.

5 Résultats

Cette section présente les premiers résultats obtenus via l'utilisation de notre prototype. Tout d'abord, nous avons effectué de nouveau le test décrit lors de la section 3.1. Cette expérience a eu pour but l'analyse du comportement et du gain apporté par le système aIOLi dans le cadre d'accès collectifs à la même ressource. Une deuxième expérience a consisté à comparer notre solution avec les 4 niveaux d'optimisations disponibles au sein du standard MPI I/O, [29]. L'ensemble des tests a été réalisé sous les mêmes conditions que dans la section 3.1. Les résultats sont plus qu'encourageant.

5.1 Décomposition de fichiers via l'appel `read`

L'expérience suivante ne se différencie de celle abordée auparavant que par le fait qu'elle ait été recompilée avec la souche cliente du service aIOLi. Les résultats sont donnés par la figure 4. Les quatre premières courbes correspondent comme précédemment à l'exécution du test avec 1, 2, 4 et 8 instances. Le premier aspect que nous souhaitons aborder est le surcoût engendré par notre solution lorsqu'une seule instance accède à une même ressource. La comparaison des courbes 1 des figures 1 et 4 (cf. page 6) permet d'observer un excédent de 30% (pour les accès de 1Ko) qui décroit pour devenir transparent à partir de 16Ko. Ce surcoût est lié aux copies intempestives entre la zone de mémoire partagée et les tampons finaux du programme test. La décomposition d'un fichier entre plusieurs processus s'avère être nettement plus efficace et ce même pour les accès de faible taille (agrégés en de plus conséquents). Le second point concerne l'exécution du test sur 2 instances. En effet, les résultats obtenus se sont révélés fort intéressant. Une analyse nous a permis de détecter que seules quelques requêtes sont agrégées/réordonnancées pour des granularités comprises entre 16Ko et 512Ko, le pire étant atteint à 128Ko où aucune optimisation n'est réalisée. Ces cas particuliers sont dues à l'ordonnanceur du noyau, au client NFS et à la taille des accès (la file d'attente des accès ne contenant jamais plus d'une demande). En s'appuyant sur ce constat, nous avons commencé à améliorer notre modèle afin de détecter au mieux les schémas d'accès parallèles intégrant une nombre restreint de clients. Actuellement, plus il y a de processus participant, plus la solution tend vers les performances maximales du système de fichiers[15].

[14] Les mécanismes de recouvrement nécessitent des opérations coûteuses de type "read modified write" afin d'assurer la cohérence ; nous avons donc volontairement choisi de ne pas les implémenter.

[15] La lecture séquentielle par un unique processus avec un grain d'accès conséquent fournit cette limitation pratique.

5.2 MPI I/O vs aIOLi

Notre dernière expérience consiste à comparer les optimisations présentes dans le modèle MPI I/O par rapport au système aIOLi. Implanté avec MPICH 1.2.5. et ROMIO, le programme test fournit 6 manières de réaliser une décomposition : l'approche commune POSIX, les 4 niveaux du standard MPI I/O (définis dans [29] et incluant les techniques "Data Sieving" (level 2) et "Two Phases" (level 3)) et enfin l'approche POSIX aIOLi. La granularité maximale d'accès (4Mo) a été fixée afin que ROMIO ne soit pas pénalisée lors des processus d'optimisations. En effet, ROMIO utilise des tampons d'agrégations/recouvrements de 4Mo par processus alors que le système aIOLI est configuré avec des segments de 32Mo en zone partagée. L'expérience étant réalisée avec 8 instances, les deux solutions peuvent être comparées (elles ont la possibilité d'agréger/recouvrir sur une taille globale identique). La figure 5 présente les résultats.

Les fonctionnalités basiques du modèle MPI I/O (niveau 0) sont simplement des routines MPI interfaçant les appels standard et ne fournissent par conséquent aucune amélioration par rapport à l'approche POSIX. Le niveau 1, processus collectif sans utilisation de masques, s'avère être le plus efficace pour des accès de la taille proche des paramètres du système de fichiers (dans notre cas 32Ko). Cela s'explique par le fait que les requêtes contiguës traitées par le noyau parallèlement (8 requêtes) s'étendent sur une plage d'offset bornée ($nb\_instance_mpi * granularite$, dans notre cas 256Ko) et profite du cache et des techniques "read ahead" coté serveur. Le niveau 2, "Data Sieving" n'est pas réellement conçu pour les accès incluant plusieurs processus : chaque processus réalise un seul appel recouvrant la totalité du fichier (grâce à l'utilisation d'un schéma d'accès défini préalablement. Cet appel est alors divisé par bloc de 4Mo (soit 256 requêtes). Le comportement est similaire à celui analysé en 3.1 où le noyau traite en parallèle 8 requêtes de 4Mo sans tenir compte de l'impact mutuel de chacune des demandes. Le niveau 3, "Two Phases", devrait fournir les meilleures performances. Néanmoins même si l'algorithme permet d'obtenir un meilleur ordonnancement des requêtes, la manière de gérer les demandes dans le noyau SMP et les caractéristiques du client NFS limitent, là encore, le gain susceptible d'être apporté (le noyau exécutant toujours en parallèle 8 requêtes non contiguës). De plus, la mise en œuvre de cette méthode demande un nombre important de transmission de messages en plus de copies intempestives entre les différentes couches ce qui entraine un surcoût significatif pour les accès de petites tailles (au moins jusqu'à 512Ko). Comme le graphique le montre, la solution aIOLi procure les meilleurs résultats : pour chaque exécution, après une brève période de transition, seul de larges requêtes sont délivrées de manière séquentielle au système de fichiers.

6 Travaux dans le contexte

Comme nous l'avons mentionné dans la partie 2, de nombreux travaux ont été réalisés dans la gestion des Entrées/Sorties Parallèles mais aucun ne répond aux exigences fixées. Certains projets continuent à travailler sur le standard MPI I/O et ses implantations. [6] suggère par exemple une nouvelle approche dans la gestion des accès non contigus. Intitulé "list I/O", cette technique est similaire au concept de "stream based I/O" développé dans PVFS ou encore l'appel "lio_listio" définit dans le standard POSIX ; elle consiste à exploiter une liste de tuples offset/taille représentant plusieurs demandes disjointes. Plus récemment, [30] a proposé plusieurs améliorations permettant de réduire le surcoût dans les techniques à base de masques de fichiers (il exploite de nouvelles fonctionnalités de l'interface MPI). [12] fournit une solution intégrant une API permettant de contrôler plus finement la correspondance entre vues logiques et vues physiques grâce à des concepts empruntés des bases de données (utilisation des vues). Cependant, l'ensemble de ces travaux requiert des API spécifiques en plus d'impliquer une connaissance rigoureuse des mécanismes internes.

Par ailleurs, des solutions proches du matériel ont également été présentées. Le système "read2", [10], exploite des caractéristiques des cartes Myrinet afin d'accéder directement aux contrôleurs disques distants et améliorer, ainsi, considérablement les performances. De même, [31] propose d'intégrer les concepts d'Entrées/Sorties Parallèles au sein des contrôleurs RAID. Ces solutions bas niveaux, en plus de nécessiter d'équipement onéreux, sont fortement intrusives et dépendantes.

7 Conclusion et travaux futures

Le système aIOLi présente des améliorations pouvant être apportées au module de gestion "classique" des Entrées/Sorties distantes dans les noyaux "multi-threaded". Au lieu de traiter chaque demande de manière parallèle et indépendante, notre solution consiste à stocker temporairement les requêtes afin de déterminer un meilleur ordonnancement et des optimisations permettant de réaliser une exploitation plus fine des serveurs distants. En intervenant au point de concentration, notre modèle permet de mettre en œuvre des techniques connues des Entrées/Sorties Parallèles sans nécessiter de mécanismes de synchronisations ou de routines sophistiquées. Dans le but de valider notre approche, nous avons présenté l'implantation d'un prototype, aIOLi, surchargeant les appels standards POSIX : `open` / `lseek` / `read` / `write` / `close`. Les résultats préliminaires observés sont encourageant. Nous avons commencé à analyser notre prototype sur des architectures 64 bits (SMPs à 16 processeurs fournis par le laboratoire BULL-HPC à Echirolles) avec le

système de fichiers Lustre. Par ailleurs, plusieurs points sont en cours d'étude afin d'améliorer le système : l'intégration de mécanismes permettant de prendre en considérations les problèmes liés aux techniques de répartitions de données sur plusieurs serveurs de stockages. De plus, nous souhaiterions proposer la solution au sein d'un module noyau de l'architecture Linux dans le but d'éliminer les surcoûts provenant d'une part des copies intempestives entre les couches et d'autre part à l'utilisation de mécanismes IPCs.

Enfin, nous avons également commencé à étudier les aspects distribués d'une telle architecture afin de déterminer les difficultés (échelle de temps, cohérence des accès, ...). L'objectif éventuel consisterait à mettre en place les principes de la solution aIOLi à chaque point de concentration[16] présents dans les grappes et les grilles (switch, router, ...).

Références

[1] MPI I/O tutorial (website). http ://hpcf.nersc.gov/software/libs/io/mpiio.html.

[2] ROMIO (website). http ://www-unix.mcs.anl.gov/romio/.

[3] 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.

[4] Gene Myron 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.

[5] 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.

[6] Avery Ching, Alok Choudhary, Kenin Coloma, Wei keng Liao (Northwestern University), Robert Ross, and William Gropp (Argonne National Laboratory). Noncontiguous i/o accesses through mpi-io. May 2003.

[7] 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.

[8] 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.

[9] 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.

[10] 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.

[11] 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.

[12] Walter F. Tichy Florin Isaila. View i/o : Improving the performance of non-contiguous i/o. In *IEEE International Conference on Cluster Computing*, Honk Kong, December 2003. IEEE Computer Society Press.

[13] John L. Hennessy, David Goldberg, and David A. Patterson. Computer architecture : A quantitative approach, 1996. Second Edition, Morgan Kaufmann Publishers, Inc.,San Francisco, California.

[14] James V. Huber, Jr., Christopher L. Elford, Daniel A. Reed, Andrew A. Chien, and David S. Blumenthal. PPFS : A high performance portable parallel file system. In Hai Jin, Toni Cortes, and Rajkumar Buyya, editors, *High Performance Mass Storage and Parallel I/O : Technologies and Applications*, chapter 22, pages 330–343. IEEE Computer Society Press and Wiley, New York, NY, 2001.

[15] David Kotz. Disk-directed I/O for MIMD multiprocessors. *Bulletin of the IEEE Technical Committee on Operating Systems and Application Environments*, pages 29–42, Autumn 1994.

[16] Comme l'est le noyau pour les SMPs

[16] 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.

[17] Pierre Lombard and Yves Denneulin. nfsp : A distributed nfs server for cluster of workstations. In *Proceeding of the 16th international Parallel and Distributed Processing Symposium*, April 2002.

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

[19] 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.

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

[21] 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.

[22] 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.

[23] 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.

[24] 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.

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

[26] 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.

[27] 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.

[28] 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.

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

[30] Joachim Worringen, Jesper Larson Traff, and Hubert Ritzdorf. Fast parallel non-contiguous file access. In *Proceedings of SC2003 : High Performance Networking and Computing*, Phoenix, AZ, November 2003. IEEE Computer Society Press.

[31] Xinrong Zhou and Tong Wei. A greedy I/O scheduling method in the storage system of clusters. In *Workshop on Parallel I/O in Cluster Computing and Computational Grids*, pages 712–717, Tokyo, May 2003. IEEE Computer Society Press. Organized at the IEEE/ACM International Symposium on Cluster Computing and the Grid 2003.

Unité de recherche INRIA Rhône-Alpes
655, avenue de l'Europe - 38334 Montbonnot Saint-Ismier (France)

Unité de recherche INRIA Futurs : Parc Club Orsay Université - ZAC des Vignes
4, rue Jacques Monod - 91893 ORSAY Cedex (France)
Unité de recherche INRIA Lorraine : LORIA, Technopôle de Nancy-Brabois - Campus scientifique
615, rue du Jardin Botanique - BP 101 - 54602 Villers-lès-Nancy Cedex (France)
Unité de recherche INRIA Rennes : IRISA, Campus universitaire de Beaulieu - 35042 Rennes Cedex (France)
Unité de recherche INRIA Rocquencourt : Domaine de Voluceau - Rocquencourt - BP 105 - 78153 Le Chesnay Cedex (France)
Unité de recherche INRIA Sophia Antipolis : 2004, route des Lucioles - BP 93 - 06902 Sophia Antipolis Cedex (France)

Éditeur
INRIA - Domaine de Voluceau - Rocquencourt, BP 105 - 78153 Le Chesnay Cedex (France)
http://www.inria.fr
ISSN 0249-6399