Un support efficace pour l'atomicité MPI basé sur le versionnage des données

Viet-Trung Tran

RenPar'20 / SympA'14 / CFSE 8
Saint-Malo, France, du 10 au 13 mai 2011

Un support efficace pour l'atomicité MPI basé sur le *versionnage* des données

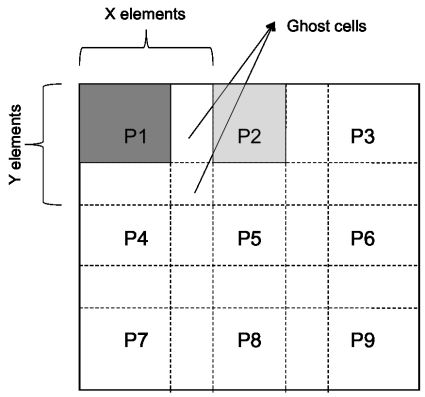
Viet-Trung Tran*, Bogdan Nicolae+
* ENS Cachan, IRISA, Rennes, France
+ INRIA Saclay-Île de France, Orsay, France

Résumé
Nous considérons le défi de la construction des systèmes de gestion des données qui répondent aux besoins des applications de calcul à hautes performances : fournir un haut débit d'E/S tout en assurant des accès simultanés aux données. Dans ce contexte, de nombreuses applications s'appuient sur MPI-IO et nécessitent l'atomicité des opérations non contiguës d'E/S pour manipuler l'accès aux données partagées. Dans la plupart des implémentations existantes, l'atomicité de l'opération est souvent mise en oeuvre en se basant sur des schémas de verrouillage qui se sont avérés inefficaces, surtout pour les E/S non contiguës. Nous affirmons que l'usage d'une couche de stockage des données à base de *versionnage* permet d'éviter la synchronisation présente dans les techniques basées sur le verrouillage, donc il est beaucoup plus efficace. Nous décrivons un prototype pour une telle couche de stockage intégrée avec ROMIO. Nous présentons les résultats de plusieurs expériences avec des programmes de test MPI-IO spécialement conçus pour évaluer les performances des accès d'E/S non contiguës qui se chevauchent sous la garantie d'atomicité de MPI.

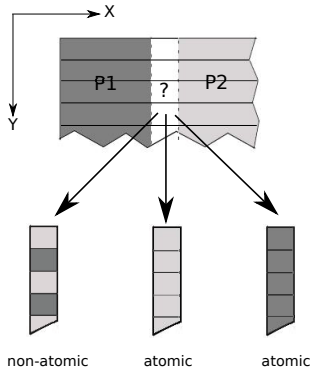
Mots-clés : grande échelle ; stockage ; atomicité de MPI-IO ; E/S non contiguës ; *versionnage*.

1. Introduction

Les applications scientifiques deviennent de plus en plus intensives en données : les simulations à haute résolution de phénomènes naturels, la modélisation du climat, l'analyse d'images à grande échelle, etc. Actuellement ces applications manipulent des volumes de données de l'ordre du pétaoctet. Dans ce contexte, il a été montré que la performance des E/S (entrées-sorties) devient rapidement problématique, engendrant un goulot d'étranglement qui a un impact négatif sur les performances globales des applications. Ce type de problème vient notamment du fait que les modèles d'accès utilisés par de telles applications scientifiques ne correspondent pas aux interfaces d'accès proposées par les systèmes de fichiers utilisés pour le stockage.
Un défi particulièrement difficile dans ce contexte est la nécessité de répondre efficacement aux besoins en termes d'E/S des applications scientifiques [13, 14], qui divisent des domaines multidimensionnels en sous-domaines se chevauchant et devant être traités en parallèle, puis stockés dans un fichier globalement partagé. La figure 1(a) représente un exemple d'un espace 2D partitionné en 3×3 sous-domaines qui se chevauchent, chacun étant géré par l'un des processus $P_1 \ldots P_9$.
Comme le fichier est une séquence plate d'octets, les sous-domaines forment des régions non-contiguës dans le fichier. Le chevauchement de telles régions non-contiguës est une source potentielle d'incohérence si les accès non-contiguës ne sont pas regroupés en transactions atomiques. Un tel état incohérent est représenté sur la figure 1(b), où deux processus MPI, P_1 et P_2, écrivent simultanément leurs sous-domaines respectifs. Seuls deux états corrects sont possibles, dans lesquels toutes les régions non-contiguës de P_1 et P_2 sont écrites de manière atomique dans le fichier. Par conséquent, *l'atomicité* des lectures et des écritures *non-contiguës et se chevauchant* à partir d'un fichier partagé est une question cruciale.
Dans un effort pour combler le fossé entre ce qui est nécessaire au niveau de l'application et ce qui est offert au niveau du système de stockage, plusieurs tentatives de normalisation ont été faites. MPI-IO [15]



(a) Un tableau 2D est divisé en sous-domaines qui se chevauchent aux frontières



(b) Un exemple de deux écritures concurrentes qui se chevauchent

FIGURE 1 – Description du problème : le partitionnement d'un domaine spatial en sous-domaines chevauchants et le problème de cohérence

fournit une interface standard pour que des programmes MPI accèdent au stockage. Pourtant, la mise en place d'opérations atomiques au sein des implémentations MPI-IO courantes, comme ROMIO [15], n'est pas si efficace. En effet le système de fichiers utilise des modèles d'accès moins puissants en pratique, comme la sémantique POSIX [1], qui force les implémentations MPI-IO à utiliser des techniques inefficaces basées sur le verrouillage pour garantir l'atomicité des accès non-contigus et se chevauchants.
Dans ce papier, notre objectif consiste à répondre aux limitations des approches existantes en optimisant le système de stockage en vue notamment des modèles d'accès décrits précédemment. Nous proposons un nouveau schéma basé sur *le versionnage* pour garantir l'atomicité des opérations. Ce schéma offre une meilleure isolation et évite la nécessité d'effectuer des synchronisations coûteuses en utilisant plusieurs *snapshots* des mêmes données. Ces *snapshots* offrent une vision globalement cohérente du fichier grâce à une gestion efficace des opérations se chevauchants au niveau des métadonnées, qui permet d'obtenir des débits élevés en présence d'accès concurrents tout en garantissant l'atomicité.

2. Travaux liés

De précédents travaux, ont montrer que fournir l'atomicité au sens de MPI de manière suffisamment efficace n'est pas une tâche triviale en pratique, surtout lorsqu'il s'agit de gérer des E/S concurrentes, non-contiguës. Plusieurs approches ont été proposées à différents niveaux : au niveau de la couche MPI-IO et au niveau du système de fichiers.
Une première série d'approches ne suppose aucun support spécifique au niveau du système de fichiers parallèle. C'est typiquement le cas de PVFS, un système de fichiers parallèle largement utilisé qui permet un accès haute performance aux données pour les opérations d'E/S tant contiguës que non-contiguës sans garantir d'atomicité. Pour les applications où l'exigence de l'atomicité MPI doit être satisfaite, une solution (par exemple illustrée dans [9]) garantit cette atomicité indépendamment du système de fichiers, au niveau de la couche MPI-IO. Généralement, l'ensemble du fichier est verrouillé pour chaque demande d'E/S et les accès concurrents sont serialisés, ce qui surcharge le système. Pour éviter ce goulot d'étranglement dans le cas d'accès non-chevauchants simultanés au même fichier partagé, une approche alternative [12] propose d'introduire un mécanisme de détection automatique des E/S non-chevauchantes et donc éviter un verrouillage dans ce cas-là. Toutefois, comme l'admettent les auteurs de cette approche, cela implique une surcharge inutile en raison du mécanisme de détection des E/S non-chevauchantes simultanées.
D'autres optimisations sont proposées dans [3], où les auteurs proposent un schéma de verrouillage pour un accès non-contigu qui vise à réduire strictement la cadre de la région verrouillée aux zones qui sont vraiment accédées. Cependant, cette approche ne permet pas d'éviter la serialisation pour les E/S

chevauchantes et simultanée comme celles décrites ci-dessus.
Dans [6], les auteurs proposent d'utiliser un processus de *handshaking* pour éviter/réduire la serialisation entre processus. Cette approche permet aux processus de négocier les uns avec les autres qui a le droit d'écrire dans les régions chevauchantes. Toutefois, une telle approche peut seulement être appliquée lorsque tous les processus ont connaissance des accès des autres processus sur le fichier. Cela n'est pas adapté pour les opérations d'E/S concurrentes non-collectives, où une telle hypothèse ne tient pas (les E/S concurrentes ne sont généralement pas conscientes les unes des autres dans ce cas-là).
Une autre classe d'approches traite le cas où le système de fichiers parallèle supporte l'atomicité POSIX. Par exemple, les systèmes de fichiers parallèles tels que GPFS [10] et Lustre [11] fournissent une sémantique d'atomicité POSIX en utilisant une approche basée sur le verrouillage distribué : les verrous sont stockés et gérés sur les serveurs de stockage qui hébergent les objets. Bien que l'atomicité POSIX peut être facilement et directement mise à profit pour les opérations E/S contiguës, l'atomicité des E/S *non-contiguës* ne peut pas efficacement se baser sur l'atomicité POSIX. Dans le schéma par défaut, si l'on considère un ensemble de chaînes d'octets non-contiguës auquel accéder en une seule requête, il est alors nécessaire de verrouiller la plus petite chaîne d'octets contiguës qui couvre tous les éléments de l'ensemble des chaînes. Cela mène à une synchronisation inutile, puisque cette technique couvre également des données non-nécessaires qui n'aurait pas besoin d'être verrouillées.

3. L'approche principale

Nous proposons une approche générale qui permet un débit élevé en présence d'accès concurrents sous la garantie d'une atomicité au sens de MPI pour les écritures dans des régions non-contiguës, chevauchantes. Cette approche repose sur trois principes de base.

Les API atomiques dédiées au niveau du stockage
Les approches traditionnelles fournissent un support des E/S atomiques non-contiguës en mettant en œuvre une interface d'accès MPI-IO au dessus d'une sémantique de cohérence standardisé (par exemple POSIX). Cette approche permet de brancher facilement un nouveau système de fichier sans avoir besoin de réécrire la couche MPI-IO. Toutefois, cet avantage a un prix élevé : la couche MPI-IO doit convertir l'atomicité MPI en des modèles de cohérence différents, ce qui limite fortement les possibilités d'optimiser le système pour les modèles d'accès dans notre contexte. A l'inverse, nous proposons d'étendre le système de stockage avec une interface d'accès aux données qui fournit un support natif des accès non-contigus, et surtout atomiques. Cette approche permet de contourner la nécessité de convertir l'atomicité MPI en un modèle de cohérence spécifique et permet d'utiliser des optimisations directes au niveau du système de stockage. Ceci permet la mise en œuvre d'un modèle plus performant de contrôle des accès concurrents.

La segmentation de données
La segmentation des données est une technique bien connue pour augmenter le débit des accès aux données, en divisant le fichier globale en *morceaux* qui sont ensuite distribués sur plusieurs fournisseurs de stockage. En utilisant une stratégie d'allocation de charge qui redirige les opérations d'écriture sur différents fournisseurs de stockage, la charge des E/S est distribuée efficacement, ce qui augmente le débit global.

Le *versionnage* pour améliorer les accès concurrents aux données
La plupart des systèmes de stockages manipulent une version unique du fichier, y compris en présence de concurrence. Ils offrent des mécanismes de verrouillage qui garantissent un accès exclusif aux régions chevauchantes, afin d'éliminer les incohérences. Cependant, une telle approche n'est pas efficace dans notre contexte, même si le serveur de stockage peut offrir un débit élevé. Le problème vient du fait qu'un système de verrouillage ajoute un surcoût, car il permet à un seul processus d'obtenir l'accès exclusif à une région à un moment donné. Comme il y a beaucoup de régions se chevauchant, cela conduit à une situation où de nombreuses écritures sont inactives et attendent leur tour, d'où une très forte limitation du débit total. Afin d'éviter ce problème, nous proposons un système d'accès basé sur le *versionnage* qui évite la nécessité des verrouillages, éliminant par la même occasion ces attentes.

4. Implémentation

Suite à l'approche proposée dans la section 3, nous avons choisi de construire un système de stockage au-dessus de *BlobSeer* [8], un service de partage de données spécifiquement conçu pour répondre aux exigences des applications intensives en données : *l'agrégation extensive d'espace de stockage* à partir d'un grand nombre de participants, le support pour *des objets de très grande taille*, *l'accès efficace à grain fin* aux données et la capacité à maintenir un *haut débit lors d'accès concurrents*.

BlobSeer considère les données comme des longues séquences d'octets appelées BLOB (*Binary Large Object*). Pour manipuler ces BLOB, BlobSeer s'appuie sur *la segmentation des données*, *la gestion distribuée des métadonnées* et *le contrôle de concurrence orienté versionnage* pour distribuer la charge de travail des E/S à grande échelle et éviter la nécessité de synchroniser l'accès tant au niveau des données que des métadonnées. Ceci est crucial pour atteindre un débit élevé en présence accès concurrents, comme il a été démontré dans [8, 7].

Le choix de construire un système de stockage au dessus de BlobSeer a été motivé par deux facteurs. Premièrement, BlobSeer divise les BLOB en pages de manière transparente. Ceci permet d'éviter la nécessité de réaliser explicitement cette segmentation des données. Deuxièmement, BlobSeer génère un nouveau *snapshot* du BLOB lors de chaque modification, tout en ne stockant physiquement que les différences entre chaque *snapshot*. Ceci fournit une base solide à l'introduction du *versionnage* en tant que principe clé de l'atomicité au sein de MPI.

Puisque BlobSeer permet uniquement l'atomicité des lectures et des écritures de régions contiguës, nous ne pouvons pas profiter directement de son interface d'accès orientée *versionnage* pour permettre l'atomicité MPI, qui doit également prendre en compte les accès non-contiguës. Des modifications ont donc dû être apportées à BlobSeer.

- Nous avons étendu l'interface de BlobSeer pour qu'il puisse décrire des accès non contigus en un seul appel, en implémentant une série de primitives orientées *versionnage* qui facilitent les accès non contigus. Ces primitives correspondent a l'interface *List I/O* introduit dans [2].
- Nous avons ajouté le support de l'atomicité pour des E/S non contiguës en étendant la technique de *versionnage* dans BlobSeer. Nous garantissons ainsi la seule publication de *snapshots* cohérents, c'est à dire respectant l'atomicité au sens de MPI.

4.1. Proposition d'une interface d'accès non-contigus orientée *versionnage*

Prenant exemple sur l' interface *List I/O* introduite dans [2], nous avons introduit deux primitives orientées *versionnage* qui facilitent les E/S de données non-contiguës au niveau des BLOB.

Un **NONCONT_WRITE(id, buffers[], offsets[], sizes[])** permet de créer un nouveau *snapshot* d'un BLOB (identifié par son id) en soumettant une liste de tampons de mémoire (buffers[]) pour écrire sur les régions non-contiguës définies par les listes offsets[] et sizes[].

De manière semblable, **NONCONT_READ(id, v, buffers[], offsets[], sizes[])** permet la lecture (récupération dans buffers[]) de régions non-contiguë d'un BLOB à une version v donnée.

4.2. Ajouter le support de l'atomicité MPI

Dans la section 3, nous avons proposé une approche basée sur le *versionnage* pour garantir efficacement l'atomicité MPI en présence d'accès concurrents. Cette approche s'appuie sur l'idée que les modifications d'un fichier par une écriture non-contiguës engendre un nouveau *snapshot* indépendant. La principale difficulté dans ce contexte est de mettre à jour efficacement les différences au niveau des métadonnées de manière à ce que seuls les *snapshots* respectant l'atomicité MPI soient publiés. Puisque chaque *snapshot* d'un BLOB est segmenté sur un grand nombre de fournisseurs de stockage, les métadonnées ont pour rôle de mémoriser l'emplacement de chaque page d'un *snapshot*, afin qu'il soit possible de faire correspondre des régions non-contiguës d'un *snapshot* avec ces pages.

BlobSeer organise les métadonnées sous la forme d'un *arbre de segments distribués* [16] : un arbre est associé à chaque *snapshot* d'un BLOB. Cet arbre consiste en un arbre binaire dans lequel chaque nœud est associé à une portée de BLOB, délimité par *offset* et *size*. La racine couvre tout le *snapshot* du BLOB, tandis que les feuilles couvrent les pages, qui sont les segments les plus fins. Afin d'éviter le surcoût de la reconstruction de l'arbre pour chaque nouveau *snapshot* (ce qui consommerais du temps et de l'espace), des sous-arbres entiers sont partagés entre les *snapshots*, comme le montre la figure 2.

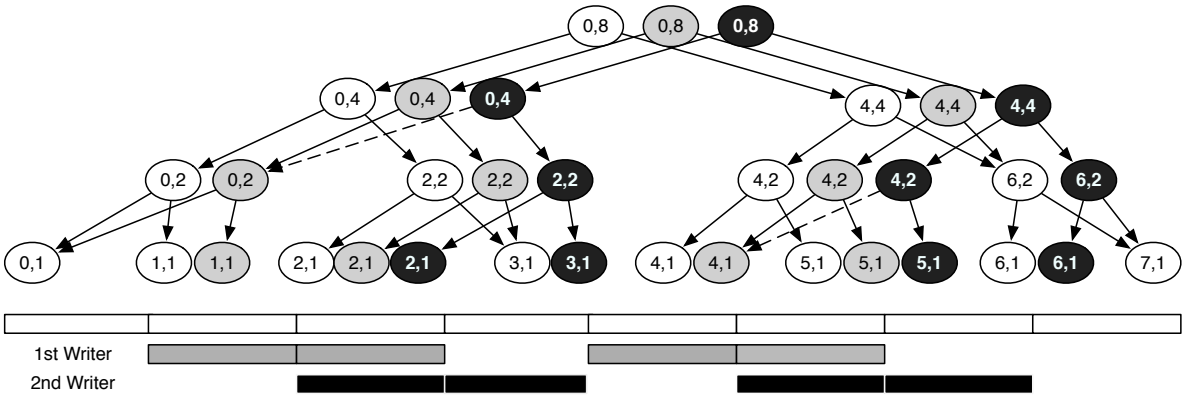


FIGURE 2 – Arbres segments de métadonnées : des sous-arbres entiers sont partagés entre les versions.

4.2.1. Écriture non-contiguë

Une opération NONCONT_WRITE envoie d'abord tous les morceaux qui composent les régions non-contiguës aux fournisseurs d'espace de stockage. Ensuite, il faut qu'elle construise l'arbre à segments des métadonnées associées au nouveau *snapshot* d'une manière ascendante : de nouvelles feuilles contenant les informations sur la localisation des nouvelles pages sont générées, puis de nouveaux noeuds internes sont créés en remontant vers la racine. Par exemple, l'écriture de deux régions non-contiguës délimitée par des couples *(offset, size)* comme $(1, 2)$ et $(4, 2)$ sur un BLOB dont la taille totale est de 8 engendre la construction de l'arbre segments gris représenté sur la Figure 2.
Dans le cas d'écritures concurrentes, les écritures simultanées peuvent s'exécuter de manière indépendante dans leurs régions non-contiguës sans avoir besoin de se synchroniser, parce que les régions non-contiguës sont classées et regroupées par *snapshots* cohérents au niveau des métadonnées. Pour éviter la synchronisation lors de la génération d'arbres de métadonnées, BlobSeer maintient la liste de toutes les écritures simultanées et l'utilise comme un renseignement pour les clients qui écrivent au moment où ces derniers demandent de créer un nouveau *snapshot*. En utilisant cette information, les clients peuvent construire leur arbre de métadonnées de manière indépendante. Dans notre exemple 2, lors de la seconde écriture (en noir) le client reçoit l'information concernant l'écriture non-contiguë précédente (en gris) de sorte qu'il puisse construire l'arbre de métadonnées noir en supposant l'existence de l'arbre gris (construit de manière indépendante). Lorsque les constructions grise et noire se terminent, les arbres segments sont dans un état cohérent.

4.2.2. Lecture non-contiguës

Une opération NONCONT_READ commence par récupérer la racine de l'arbre segment qui correspond au *snapshot* à partir duquel elle a souhaite lire. Ensuite, elle descend dans l'arbre segment vers les feuilles qui contiennent des informations sur les morceaux qui composent la région non-contiguë. Une fois que ces morceaux ont été trouvés, ils sont récupérés auprès des fournisseurs de stockage et copiés dans la mémoire tampon fourni comme argument. Étant donné que les *snapshots* déjà publiés ne sont jamais modifiés, les opérations de lecture n'ont jamais besoin d'être synchronisées avec les opérations d'écriture. Donc, la lecture non-contiguë est toujours atomique.

4.3. Connecter l'interface orientée *versionnage* au niveau de la couche MPI-IO

Ayant obtenu une implémentation d'un système de stockage directement optimisée pour l'atomicité MPI, la prochaine étape est d'utiliser de façon efficace ce stockage au niveau de la couche MPI-IO. Nous avons donc utilisé ROMIO [15], une bibliothèque qui fait partie de MPICH2 [4]. Notre prototype fournissant un support natif pour les deux types d'écritures contiguës et non-contiguës sous la garantie de l'atomicité MPI, ROMIO n'a pas besoin de convertir la sémantique d'atomicité MPI en une sémantique valide au niveau du système de fichier, ce qui aurait été une source de surcharge au niveau des performance des applications.

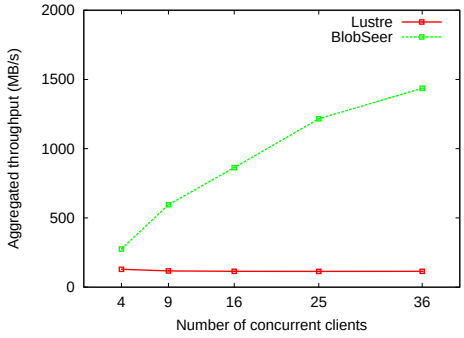


FIGURE 3 – Notre approche par rapport à l'approche basée sur le verrouillage.

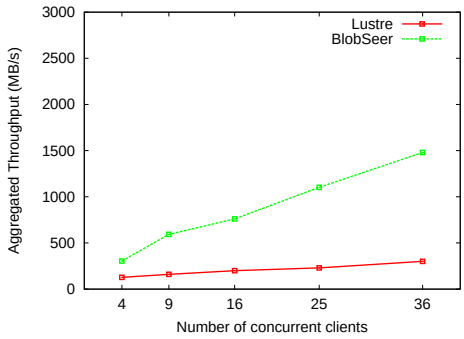


FIGURE 4 – MPI-tile-IO : 1024×1024×1024 tuiles

5. Evaluation expérimentale

5.1. Vue d'ensemble

Nous avons construit un système de stockage qui spécialement optimisé pour des accès non-contiguës, chevauchants, et garantissant l'atomicité MPI. Pour évaluer notre prototype, nous comparons notre approche à l'approche basée sur le verrouillage qui s'appuie sur un système de fichiers compatible POSIX, qui est la façon traditionnelle d'aborder l'atomicité MPI. Le système de fichiers parallèles Lustre [11] a été choisi pour cela. Lustre ainsi que notre prototype sont connectés à ROMIO grâce à leur interface ADIO respective. Le mode *data sieving* de Lustre a été désactivé selon les recommandations faites dans [3], de manière à en améliorer les performances.

Deux séries d'expériences ont été conduites. Notre première expérience vise à évaluer la scalabilité de notre approche en augmentant le nombre de clients concurrents qui écrivent sur des régions non-contiguës dans un même fichier. Dans ce scénario, nous avons considéré le cas extrême où chacun des clients écrit un grand nombre de régions non-contiguës qui sont délibérément choisies de manière à générer un grand nombre de chevauchement nécessitant l'atomicité MPI. Dans la deuxième expérience, nous avons effectué une évaluation des performances de notre approche en utilisant un test standard de performance, *MPI-tile-IO*. Ce test de performance simule les modèles d'accès des applications scientifiques réelles qui partagent les données d'entrée en sous-domaines se chevauchant dont les E/S doivent être atomiques.

Les expériences ont été effectuées sur Grid'5000 [5], une plate-forme de type grille reconfigurable, contrôlable qui rassemble 9 sites géographiquement distribués en France. Nos expériences ont été effectuées sur 80 nœuds du site de Rennes de la façon suivante : Lustre (puis BlobSeer) est déployé sur 44 nœuds tandis que les 36 nœuds restants sont réservés pour déployer un programme MPI , chaque processus MPI utilisant un noeud dédié.

5.2. Évaluer la scalabilité : notre approche par rapport à l'approche basée sur le verrouillage

Dans ce scénario, nous avons réalisé un test synthétique correspondant à un cas particulier de chevauchement des sous-domaines illustré dans la figure 1(a) : les sous-domaines qui doivent être écrits sous la garantie d'atomicité MPI forment une seule rangée. Chaque sous-domaine est une matrice de 1024×1024 éléments, chaque élément est formé de 1024 octets, engendrant un total de 1 Go de données écrites par chaque processus en 1024 régions non-contiguës, dont chacune occupe 1 Mo. Comme les sous-domaines sont disposés sur une rangée, chaque processus MPI (à l'exception des extrémités) partage une région chevauchante gauche et droite avec ses voisins. La taille de chaque région chevauchante est fixée à 128×1024 éléments. Ce choix conduit à un scénario qui pousse les deux approches à leurs limites : chaque région génère au moins un chevauchement qui doit être traité de façon atomique.

Nous avons fait varier le nombre de processus MPI entre 4 et 36 et exécuté le test MPI avec notre approche et avec l'approche basée sur le verrouillage dans Lustre. Les résultats sont présentés dans la figure 3, où l'on mesure le débit total cumulé atteint par tous les processus, soit le montant total des

données divisé par le temps d'exécution des E/S.
Comme on peut le constater, le débit total dans le cas de Lustre reste constant à 114 Mo. Puisque les processus sont disposés sur une rangée, les régions non-contiguës de chaque processus sont très éloignées, ce qui conduit à une situation dans laquelle presque tout le fichier doit être verrouillé. Ainsi, les accès sont pratiquement sérialisés dans l'approche basée sur le verrouillage. En revanche, le débit agrégé dans le cas de notre approche augmente d'environ 300 Mo/s à 1500 Mo/s pour 36 clients simultanés, cela grâce au fait que cette approche évite complètement la synchronisation.

5.3. Résultats du test MPI-tile-IO

Dans ce scénario, nous avons évalué la performance de notre approche en utilisant le test synthétique *MPI-tile-IO*. MPI-tile-IO simule fidèlement les modèles d'accès des applications scientifiques réelles décrites dans la section 1.
Dans ce test, le problème (Figure 1(a)) se compose d'un tableau de 1024×1024 éléments par processus, où chaque élément est de taille 1024 octets. Chaque processus écrit 1 Go de données, donc pour 36 processus, la taille totale des données écrite sera de 36 Go. Un nombre de 4, 9, 16, 25 et 36 processus sont disposés sur des grilles 2×2, 3×3, 4×4, 5×5 et 6×6. Le chevauchement entre processus est configuré pour une taille de 128×128 éléments. Le test MPI-IO-tile est exécuté sur la base de notre approche et sur celle de Lustre. La figure 4 représente le débit total obtenu pour les deux approches.
Comme on peut le remarquer, les deux approches passent à l'échelle. Contrairement à l'expérience présentée dans la section 5.2, le choix d'organiser les sous-domaines de telle sorte qu'ils forment un carré aboutit à une situation où une région contiguë plus compacte doit être verrouillée par l'approche fondée sur Lustre. Ceci conduit à une situation où différentes régions contiguës des fichiers peuvent être verrouillées simultanément, améliorant ainsi le degré de parallélisme. Néanmoins, même dans de telles circonstances, un grand nombre d'accès doivent être sérialisés, le débit agrégé de cette approche est encore beaucoup moins bon que celui de notre approche.

6. Conclusions

Dans cet article, nous avons proposé un mécanisme original basé sur le concept de *versionnage*. Il peut être exploité pour répondre efficacement aux besoins des applications MPI intensives en données qui présentent des E/S chevauchantes non-contiguës où l'atomicité MPI-IO doit être garantie.
Contrairement aux approches traditionnelles qui s'appuient sur les systèmes de fichiers parallèles et emploient les schémas de verrouillage pour l'atomicité MPI, nous proposons d'utiliser des techniques de *versionnage* pour atteindre des débits d'E/S élevés y compris en présence d'accès concurrents tout en garantissant l'atomicité MPI. Nous avons mis cette idée en pratique en étendant BlobSeer par une interface d'accès aux données non-contiguës. Cette interface a été directement intégrée dans ROMIO. Nous avons comparé notre prototype basé sur BlobSeer avec une approche standard basée sur le verrouillage. Concrètement, cette comparaison a été effectuée en utilisant le système de fichiers Lustre. Notre approche a montré un excellent passage à l'échelle sous la concurrence par rapport à l'approche implémentée par Lustre. Elle atteint un débit agrégé entre 3,5 fois à 10 fois plus élevé dans plusieurs expériences, comprenant les tests de performances standardisés pour MPI-IO qui sont utilisés pour mesurer les performances des opérations d'E/S atomiques, non-contiguës, et se chevauchant.
Ces résultats nous encouragent à poursuivre ces travaux. En particulier, un avantage de BlobSeer que nous n'avons pas développé dans ce article est d'exposer son interface de *versionnage* directement au niveau applicatif. La capacité d'utiliser des versions au niveau applicatif apporte plusieurs avantages potentiels, comme dans les applications du type producteur-consommateur, où la sortie des simulations est en même temps utilisée comme entrée par le processus de visualisation. L'utilisation de versions au niveau applicatif pourrait éviter la synchronisation, ce qui est un problème reconnu dans des approches actuelles.

Bibliographie

1. *Information technology - Portable Operating System Interface (POSIX)*. Institute of Electrical & Electronics Engineers, 2009.
2. Avery Ching, Alok Choudhary, Wei-keng Liao, Rob Ross, and William Gropp. Noncontiguous I/O

through PVFS. In *CLUSTER '02 : Proceedings of the IEEE International Conference on Cluster Computing*, CLUSTER '02, pages 405–, Washington, DC, USA, 2002. IEEE Computer Society.
3. Avery Ching, Wei-keng Liao, Alok Choudhary, Robert Ross, and Lee Ward. Noncontiguous locking techniques for parallel file systems. In *SC '07 : Proceedings of the 2007 ACM/IEEE Conference on Supercomputing*, pages 1–12, November 2007.
4. William Gropp, Ewing Lusk, and Anthony Skjellum. *Using MPI : Portable Parallel Programming with the Message Passing Interface*. MIT Press, 1994.
5. Yvon Jégou, Stephane Lantéri, Julien Leduc, Melab Noredine, Guillaume Mornet, Raymond Namyst, Pascale Primet, Benjamin Quetier, Olivier Richard, El-Ghazali Talbi, and Touche Iréa. Grid'5000 : a large scale and highly reconfigurable experimental grid testbed. *International Journal of High Performance Computing Applications*, 20(4) :481–494, November 2006.
6. W.-K. Liao, Alok Choudhary, K. Coloma, G.K. Thiruvathukal, L. Ward, E. Russell, and N. Pundit. Scalable implementations of MPI atomicity for concurrent overlapping I/O. In *ICPP '03 : Proceedings of International Conference on Parallel Processing*, pages 239–246, October 2003.
7. B. Nicolae, G. Antoniu, and L. Bougé. Enabling high data throughput in desktop grids through decentralized data and metadata management : The BlobSeer approach. *Euro-Par 2009 Parallel Processing*, pages 404–416, 2009.
8. Bogdan Nicolae, Gabriel Antoniu, Luc Bougé, Diana Moise, and Alexandra Carpen-Amarie. Blobseer : Next generation data management for large scale infrastructures. *Journal of Parallel and Distributed Computing*, 2010. In press.
9. R. Ross, R. Latham, W. Gropp, R. Thakur, and B. Toonen. Implementing MPI-I/O atomic mode without file system support. In *CCGRID '05 : Proceedings of the 5th IEEE International Symposium on Cluster Computing and the Grid*, CCGRID '05, pages 1135–1142, Washington, DC, USA, 2005. IEEE Computer Society.
10. Frank Schmuck and Roger Haskin. GPFS : A shared-disk file system for large computing clusters. In *FAST '02 : Proceedings of the 1st USENIX Conference on File and Storage Technologies*, FAST '02, Berkeley, CA, USA, 2002. USENIX Association.
11. P. Schwan. Lustre : Building a file system for 1000-node clusters. In *Proceedings of the Linux Symposium*, 2003.
12. Saba Sehrish, Jun Wang, and Rajeev Thakur. Conflict detection algorithm to minimize locking for MPI-I/O atomicity. In *Euro PVM/MPI '09 : Proceedings of the 16th European PVM/MPI Users' Group Meeting on Recent Advances in Parallel Virtual Machine and Message Passing Interface*, pages 143–153, Berlin, Heidelberg, 2009. Springer-Verlag.
13. E. Smirni, R.A. Aydt, A.A. Chien, and D.A. Reed. I/O requirements of scientific applications : An evolutionary view. In *HPDC '02 : Proceedings of 5th IEEE International Symposium on High Performance Distributed Computing*, pages 49–59. IEEE, 2002.
14. E. Smirni and D. A. Reed. Lessons from characterizing the I/O behavior of parallel scientific applications. *Performance Evaluation*, 33(1) :27–44, 1998.
15. Rajeev Thakur, William Gropp, and Ewing Lusk. On implementing MPI-I/O portably and with high performance. In *IOPADS '99 : Proceedings of the 6th Workshop on I/O in parallel and distributed systems*, pages 23–32, New York, NY, USA, 1999. ACM.
16. C. Zheng, G. Shen, S. Li, and S. Shenker. Distributed segment tree : Support of range query and cover query over DHT. In *IPTPS '06 : Proceedings of the 5th International Workshop on Peer-to-Peer Systems*, 2006.