HAL
open science

Efficient support for MPI-IO atomicity based on versioning

Viet-Trung Tran, Bogdan Nicolae, Gabriel Antoniu, Luc Bougé

To cite this version:

Viet-Trung Tran, Bogdan Nicolae, Gabriel Antoniu, Luc Bougé. Efficient support for MPI-IO atomicity based on versioning. [Research Report] RR-7487, INRIA. 2010, pp.24. inria-00546956

HAL Id: inria-00546956

https://inria.hal.science/inria-00546956v1

Submitted on 15 Dec 2010

HAL is a multi-disciplinary open access archive for the deposit and dissemination of scientific research documents, whether they are published or not. The documents may come from teaching and research institutions in France or abroad, or from public or private research centers.

L'archive ouverte pluridisciplinaire **HAL**, est destinée au dépôt et à la diffusion de documents scientifiques de niveau recherche, publiés ou non, émanant des établissements d'enseignement et de recherche français ou étrangers, des laboratoires publics ou privés.

INSTITUT NATIONAL DE RECHERCHE EN INFORMATIQUE ET EN AUTOMATIQUE

Efficient support for MPI-IO atomicity based on versioning

Viet-Trung Tran, Bogdan Nicolae, Gabriel Antoniu, Luc Bougé

N° 7487

Novembre 2010

Rapport de recherche

ISSN 0249-6399 ISRN INRIA/RR--7487--FR+ENG

INSTITUT NATIONAL DE RECHERCHE EN INFORMATIQUE ET EN AUTOMATIQUE

INRIA

centre de recherche
RENNES - BRETAGNE ATLANTIQUE

Efficient support for MPI-IO atomicity based on versioning

Viet-Trung Tran*, Bogdan Nicolae†, Gabriel Antoniu‡, Luc Bougé*

Thème : Calcul distribué et applications à très haute performance
Équipe-Projet KerData

Rapport de recherche n° 7487 — Novembre 2010 — 21 pages

Abstract:

We consider the challenge of building data management systems that meet an important requirement of today's data-intensive HPC applications: to provide a high I/O throughput while supporting highly concurrent data accesses. In this context, many applications rely on MPI-IO and require atomic, non-contiguous I/O operations that concurrently access shared data. In most existing implementations the atomicity requirement is often implemented through locking-based schemes, which have proven inefficient, especially for non-contiguous I/O. We claim that using a versioning-enabled storage backend has the potential to avoid expensive synchronization as exhibited by locking-based schemes, which is much more efficient. We describe a prototype implementation on top of ROMIO along this idea, and report on promising experimental results with standard MPI-IO benchmarks specifically designed to evaluate the performance of non-contiguous, overlapped I/O accesses under MPI atomicity guarantees.

Key-words: large scale; storage; MPI-IO atomicity; non-contiguous I/O; versioning;

* ENS Cachan, IRISA, France.
† University Rennes 1, IRISA, France.
‡ INRIA Rennes-Bretagne Atlantique, IRISA.

Centre de recherche INRIA Rennes – Bretagne Atlantique
IRISA, Campus universitaire de Beaulieu, 35042 Rennes Cedex
Téléphone : +33 2 99 84 71 00 — Télécopie : +33 2 99 84 71 71

Un support efficace basé sur le versioning pour l'atomicité de MPI-IO

Résumé : Nous considérons le défi de la construction des systèmes de gestion des données qui répondent a l'éxigence importante des applications de calcul haut performance: fournir un haut débit d'E/S tout en assurant des accès simultanées aux données. Dans ce contexte, de nombreuses applications s'appuient sur MPI-IO et nécessitent l'atomicité des opérations non contigus opérations d'E/S pour manipuler l'accès aux données partagées. Dans la plupart des existants implémentations, l'atomicité de l'opération est souvent mis en oeuvre en se basant sur les schémas de verrouillage, qui se sont avérées inefficaces, surtout pour les E/S non contigus. Nous affirmons que l'usage d'un versioning compatible stockage permet d'éviter la synchronisation présentée dans les techniques basées sur les verrouillages, donc il est beaucoup plus efficace. Nous décrivons un prototype d'un versioning compatible stockage qui est intégré à ROMIO, et en montrant les résultats des expérimentes avec des repères standard MPI-IO spécialement conçu pour évaluer les performances des accès d'E/S non contigus qui chevauchent sous la garantie d'atomicité de MPI.

Mots-clés : Grande échelle; stockage; l'atomicité de MPI-IO; E/S non-contigus; versioning;

Contents

1 Introduction **3**

2 Problem description **4**

3 Related work **6**

4 Design principles **7**

5 Implementation **8**
5.1 BlobSeer: towards a storage backend optimized for non-contiguous, MPI-atomic writes . . . 9
5.2 Proposal for a non-contiguous, versioning-oriented access interface 10
5.3 Adding support for MPI-atomicity . . . 11
5.3.1 Structure of metadata . . . 11
5.3.2 Non-contiguous writes . . . 12
5.3.3 Non-contiguous reads . . . 12
5.3.4 Guaranteeing MPI atomicity efficiently under concurrency 12
5.4 Leveraging our versioning-oriented interface at the level of the MPI-IO layer . . . 13

6 Experimental evaluation **14**
6.1 Overview . . . 14
6.2 Platform description . . . 15
6.3 Increasing number of non-contiguous regions . . . 15
6.4 Scalability under concurrency: our approach vs. locking-based . . 15
6.5 MPI-tile-IO benchmark results . . . 17

7 Conclusions **19**

1 Introduction

Scientific applications are becoming increasingly data-intensive: high-resolution simulations of natural phenomenons, climate modeling, large-scale image analysis, etc. Such applications currently manipulate data volumes in the petabyte scale and with the growing trend of data sizes we are rapidly advancing towards the exabyte scale. In this context, I/O performance has been repeatedly pointed out as a source of bottleneck that negatively impacts the performance of the applications.

Rapid advances in other system components are partly responsible for poor I/O performance, however, another problem that is causing this I/O bottleneck is the fact that the I/O access patterns generated by such scientific applications do not match the I/O access interfaces exposed by the file systems that are used as the underlying storage backends.

One particularly difficult challenge in this context is the need to efficiently address the I/O needs of scientific applications [1, 2, 3, 4] that partition multi-dimensional domains into overlapping subdomains that need to be processed in parallel and then stored in a globally shared file. Since the file is a flat sequence of bytes, subdomains map to non-contiguous regions in the file. Because the

subdomains overlap, under concurrent accesses such non-contiguous regions may interleave in an inconsistent fashion if they are not grouped together as a single atomic transaction. Therefore, *atomicity* of *non-contiguous, overlapping* reads and writes of data from a shared file is a crucial issue.

In an effort to abridge the gap between what is needed at application level and what is offered at the level of storage system, several standardization attempts have been made. MPI-IO [5] provides a standard interface for MPI programs to access the storage in a coordinated manner. However, MPI-IO implementations currently in circulation, such as ROMIO [5], have limited potential to be leveraged to their maximal potential. This happens because storage backends use less capable access models in practice, such as POSIX [6], that force the MPI-IO implementations to use inefficient locking-based schemes in order to guarantee atomicity for non-contiguous, overlapping accesses.

In this paper, we aim at addressing this shortcoming of existing approaches by optimizing the storage backend specifically for the access patterns described above. We propose a novel versioning-based scheme that offers better isolation and avoids the need to perform expensive synchronization by using multiple snapshots of the same data. These snapshots offer a consistent view of the globally shared file through efficient ordering and resolution of overlappings at metadata level, which enables obtaining high throughputs under concurrency while guaranteeing atomicity.

The contributions of this paper are summarized as follows:

- We introduce a set of generic design principles that leverage versioning techniques and data striping to build a storage backend that explicitly optimizes for non-contiguous, overlapped I/O accesses that obey MPI atomicity guarantees;
- We describe a prototype built along this idea, based on BlobSeer, a versioning-enabled, concurency-optimized data management service that was integrated with ROMIO;
- We report on a series of experiments performed with both custom as well as standard MPI-IO benchmarks specifically written for the applications that we target and show improvements in aggregated throughput under concurrency between 3.5 to 10 times higer than what state-of-art locking-based approaches can deliver.

2 Problem description

In a large class of scientific applications, especially large-scale simulations, input and output data represents huge spatial domains made of billions of cells associated with a set of parameters (e.g., temperature, pressure, etc.). These spatial domains represent the state of the simulated system at a specific moment in time. They are iteratively refined by the simulation in order to obtain an insight into how the system evolves in time. The large size of the spatial domains triggers the need to parallelize the simulation, by splitting them into subdomains that are distributed and processed by a large number of compute elements, typically MPI processes.

In many such simulations, the contents of a cell depends on the contents of the neighboring cells. Thus, the cells at the border of a subdomain (called

"ghost cells") need to be shared by more than one MPI process. In order to avoid repeated exchanges of border cells between MPI processes during the simulation, a large class of applications [1, 2, 3, 4] partition the spatial domain in such way that the resulting subdomains overlap at their borders. Figure 1(a) depicts an example of a 2D space partitioned in 3×3 overlapped subdomains, each being handled by one of the processes $P_1 \dots P_9$.

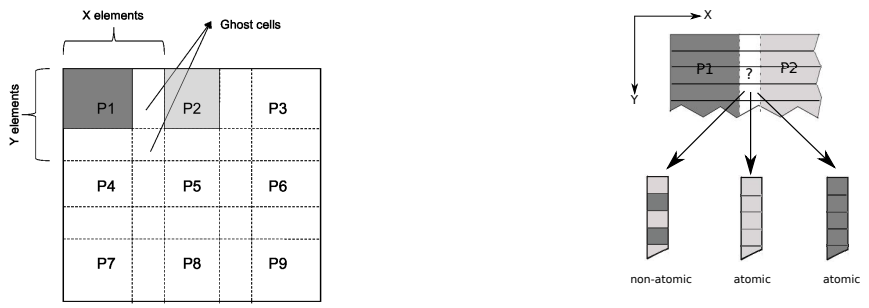
At each iteration, the MPI processes typically dump their subdomains in parallel in a globally shared file, which is then later used to interpret and/or visualize the results. Since the spatial domain is a multidimensional structure which is stored as a single, flat sequence of bytes, the data corresponding to each subdomain maps to a set of non-contiguous regions within the file. This in turn translates to a write-intensive access pattern where the MPI processes *concurrently write a set of non-contiguous regions in the same file.* Moreover, because some subdomains overlap, the non-contiguous regions in the file belonging to such subdomains may overlap too.

In order to obtain a globally shared file that represents the whole spatial domain in a consistent way, it is important to write all non-contiguous regions belonging to the same subdomain in an atomic fashion. This is a non-trivial issue, because of the poor implementations of storage backends. Either they do not offer guarantees with respect to atomicity, which means concurrent writes need to be coordinated at application level. Or they offer limited guarantees, such as the POSIX atomicity semantics [6], where a write of a contiguous region is guaranteed to end up as a single sequence of bytes, without interleaving with other writes. However, in our context, the POSIX atomicity guarantees are not sufficient, because a single write operation involves a whole set of non-contiguous regions. Under concurrency, this can lead to an interleaved overlapping that generates inconsistent states. Such an inconsistent state is depicted on Figure 1(b), where two MPI processes, P_1 and P_2, concurrently write their respective subdomains. Only two consistent states are possible, where all the non-contiguous regions of P_1 and P_2 are atomically written into the file. They only differ by the order in which this happened, but each of them correspond to some serialized behavior.

In an effort to standardize such access patterns, the MPI 2.0 standard [5] defines a specialized I/O interface, called MPI-IO, that enables read and write primitives to accept complex data types as parameters. These data types can represent a whole set of non-contiguous regions rather than a single contiguous region, as is the case of the POSIX primitives.

Under these circumstances, atomicity guarantees for write operations that involce multiple non-contiguous overlapping regions are needed. This type of atomicity is referred to as *MPI atomicity.* More precisely, the *MPI atomicity* is defined as a guarantee that in concurrent, overlapping MPI-IO write operations (which can possibly involve sets of non-contigous regions), the overlapped regions shall contain data only from one of the MPI processes that participates in the concurrent writes.

It is now clear that a write support mechanism that achieves high throughput under concurrency and introduces support for *non-contiguous, overlapping* regions while obeying *MPI atomicity semantics* is a crucial issue. Such a mechanism facilitates an efficient implementation of the MPI-IO standard, which in turn benefits a large class of MPI applications.



(a) 2D array partitioning with overlapping at the border

(b) An example of two concurrent overlapping writes

Figure 1: Problem description: partitioning of spatial domains into overlapped subdomains and the resulting I/O access patterns and consistency issues

3 Related work

Previous work has shown that providing MPI atomicity efficiently enough is not a trivial task in practice, especially when dealing with concurrent, non-contiguous I/O, most of which rely on locking-based techniques. Several approaches have been proposed at various levels: at the level of the MPI-IO layer, at the level of the file system and at the application level.

A first series of approaches assumes no specific support at the level of the parallel file system This is typically the case of PVFS [7], a widely-used parallel file system which makes the choice of enabling high-performance data access for both contiguous and non-contiguous I/O operations without guaranteeing atomicity at all for I/O operations. For applications where MPI atomicity requirement needs to satisfied, a solution (e.g. illustrated in [8]) consists in guaranteeing MPI atomicity portably at the level of the MPI-IO layer. The lack of guarantees on the semantics of I/O operations provided by the file system comes however at a high cost introduced by the use of coarse-grain locking at a higher level. Typically, the whole file is locked for each I/O request and thus concurrent accesses are serialized, which is an obvious source of overhead. To avoid this bottleneck in the case of concurrent non-overlapping accesses to the same shared file, an alternative approach [9] proposes to introduce a mechanism for automatic detection of overlapping I/O and thus avoid locking in this case. However, as acknowledged by the authors of this approach, an unnecessary overhead due to the detection mechanism is then introduced for non-overlapping concurrent I/O.

Further optimizations are proposed in [10], where the authors propose a locking-based scheme for non-contiguous I/O which aims to strictly reduce the scope of the locked regions to the areas that are actually accessed. Moreover, this approach cannot avoid serialization for applications which exhibit concurrent overlapping I/O such as the ones described in Section 2.

In [11], the authors propose to use process handshaking to avoid/reduce interprocess serialization. This approach enables processes to negotiate with each other who has the right to write to the overlapped regions. However, such an approach can only be applied when every process is aware of all other concurrent

processes accessing the same file. This is not suitable for non-collective concurrent I/O operations, where such an assumption does not hold (concurrent I/O requests are typically not aware of each other in this case).

Another class of approches addresses the case where the underlying parallel file system supports POSIX atomicity. Atomic contiguous I/O can then seamlessly be mapped to atomic read/write primitives provided by the POSIX interface. However, POSIX atomicity alone is not enough to provide the necessary atomicity guarantees for applications that exhibit concurrent, non-contiguous I/O operations. It is important to note that, to enable MPI atomicity, both contiguous and non-contiguous I/O requests need to be considered.

Parallel file systems such as GPFS [12] and Lustre [13] provide POSIX atomicity semantics using a distributed locking approach: locks are stored and managed on the storage servers hosting the objects they control. Whereas POSIX atomicity can simply and directly be leveraged for contiguous I/O operations using *byte range locking*, enabling atomic *non-contiguous* I/O based on POSIX atomicity is not efficient. In the default scheme, if we consider a set of non-contiguous byte ranges to be atomically accessed by an individual I/O request, it is then necessary to lock the smallest contiguous byte range that covers all elements of the set of ranges to be accessed. This leads to unnecessary synchronization and thus to a potential bottleneck, since this contiguous byte range also covers unaccessed data that would not need to be locked.

Finally, given the limitations of the approaches described above, an ultimate solution is to design the parallel application in such a way that MPI atomicity is not required, e.g. by enabling each process of the parallel application to write to a separate file at every iteration, then manage this set of files with custom post-processing tools. The CM1 [14] tornado simulation illustrates this approach.

In this paper we focus on data-intensive applications that require data partitioning schemes that exhibit overlapping concurrent I/O,which need MPI-IO atomicity to be guaranteed. As explained above, MPI-IO atomicity has mainly been enabled using locking-based approaches. In this paper, we propose a novel versioning-based mechanism allowing to handle atomic contiguous *and non-contiguous* I/O more efficiently compared to previous lock-based solutions.

4 Design principles

We propose a general approach to solve the issue of enabling a high throughput under concurrency for writes of non-contiguous, overlapped regions under MPI atomicity guarantees. This approach relies on three key design principles:

Dedicated API at the level of the storage backend Traditional approaches address the issue of providing support for MPI atomic, non-contiguous writes by implementing the MPI-IO access interface as a layer on top of highly standardized consistency semantics models (e.g. POSIX). The rationale behind this approach is to be able to easily plug in a different storage backend without the need to rewrite the MPI-IO layer. However, this advantage comes at a high price: the MPI-IO layer needs to translate the MPI atomicity into a different consistency model, which greatly limits the potential to optimize for the access patterns that are needed in our context. By contrast, we propose to extend the

storage backend with a data access interface that provides native support for non-contiguous, MPI-atomic data accesses. Using this approach circumvents the need to translate to a different consistency model and enables introducing optimizations directly at the level of the storage backend, enhancing the potential to implement a a better concurrency control scheme.

Data striping Simulations are becoming increasingly complex, processing huge spatial domains that easily reach the petabyte-scale order. This increasing trend of data sizes reflects not only globally, but also on the data sizes that need to be handled by each process individually. As a general rule, the computation to I/O ratio is steadily decreasing, which means that the performance of the whole application depends more and more on the performance of the I/O. In this context, a centralized solution clearly does not scale. Data striping a well-known technique to increase the throughput of data accesses, by splitting the huge file where the spatial domain is stored into *chunks* that distributed among multiple storage elements. Using a load-balancing allocation strategy that redirects write operations to different storage elements in a round robin fashion, the I/O workload is distributed by itself, which effectively increases the overall throughput that can be achieved.

Versioning as a key to enhance data access under concurrency Most storage backends manipulate a single version of the file at a time under concurrency by providing locking-based mechanisms that guarantee exclusive access to overlapped regions, in order to eliminate inconsistencies. However, in our context, such an approach does not scale, even if the storage backend is able to deliver high throughputs. The problem comes from the fact that a locking-based scheme is expensive, as it enables only one writer at a time to gain exclusive access to a region. Since there are many overlapped regions, this leads to a situation where many writers sit idle while waiting for their turn to lock, which greatly limits the potential to achieve a globally high aggregated throughput. In order to avoid this issue, we propose a versioning-based access scheme that avoids the need to lock, effectively eliminating the need of writers to wait for each other. Our approach is based on *shadowing* techniques [15], which means to offer the illusion of creating a new standalone snapshot of the file for each update to it but to physically store only the differences and manipulate metadata in such way that the aforementioned illusion is upheld. Starting from the principles introduced in [16], we propose to enable concurrent MPI processes to write their non-contiguous regions in complete isolation, without having to care about overlappings and synchronization, which is made possible by keeping data immutable: new differences are added rather than modify an existing snapshot. It is at the metadata level where the ordering is done and the overlappings are resolved in such way as to expose a snapshot of the file that looks as if all differences were applied in an arbitrary sequential order. A concrete proposal of how to achieve this in practice is detailed in Section 5.

5 Implementation

Based on the design principles introduced in the previous section, in this section we present the work we did to extend a versioning-oriented data sharing service

such that it efficiently exposes an MPI atomic, non-contiguous versioning access interface. We then show how to integrate this work with an existing MPI-IO middleware to obtain a fully functional and efficient storage solution for MPI applications.

5.1 BlobSeer: towards a storage backend optimized for non-contiguous, MPI-atomic writes

We have chosen to build the storage backend on top of *BlobSeer*, a versioning-oriented data sharing service specifically designed to meet the requirements of data-intensive applications that are distributed at large scale: *scalable aggregation of storage space* from a large number of participating machines with minimal overhead, support to store *huge data objects*, *efficient fine-grain access* to data subsets and ability to sustain a *high throughput under heavy access concurrency*.

Data is abstracted in BlobSeer as long sequences of bytes called BLOBs (Binary Large OBject). These BLOBs are manipulated through a simple access interface that enables creating a blob, reading/writing a range of *size* bytes from/to the BLOB starting at a specified *offset* and appending a sequence of *size* bytes to the BLOB. This access interface is designed to support shadowing explicitly: each time a write or append is performed by the client, a new snapshot of the BLOB is generated that acts as a first class object rather than overwriting any existing data (but physically stored is only the difference). This snapshot is labeled with an incremental version and the client is allowed to read from any past snapshot of the BLOB by specifying its version.

BlobSeer relies on *data striping*, *distributed metadata management* and *versioning based concurrency control* to distribute the I/O workload at large-scale and avoid the need for access synchronization both at data and metadata level. This is crucial in achieving a high aggregated throughput under concurrency, as demonstrated in [16, 17, 18].

This is achieved by orchestrating a series of distributed communicating processes, whose role is detailed below.

- *Data (storage) providers* physically store the chunks generated by appends and writes. New data providers may dynamically join and leave the system.
- *The provider manager* keeps information about the available storage space and schedules the placement of newly generated chunks. It employs a configurable chunk distribution strategy to maximize the data distribution benefits with respect to the needs of the application.
- *Metadata (storage) providers* physically store the metadata that allows identifying the chunks that make up a snapshot version. A distributed metadata management scheme is employed to enhance concurrent access to metadata.
- *The version manager* is in charge of assigning new snapshot version numbers to writers and appenders and to reveal these new snapshots to readers. It is done so as to offer the illusion of instant snapshot generation, while guarateeing total ordering and atomicity.

Motivation The choice of building the storage backend on top of BlobSeer was motivated by two factors.

First, BlobSeer supports transparent striping of BLOBs into chunks and enables fine-grain access to them, which enables to store each spatial domain directly as a BLOB. This in turn avoids the need to perform data striping explicitly.

Second, BlobSeer offers out-of-the-box support for shadowing by generating a new BLOB snapshot for each fine-grain update while physically storing only the differences. This provides a solid foundation to introduce versioning as a key principle to support MPI atomicity.

5.2 Proposal for a non-contiguous, versioning-oriented access interface

The versioning-oriented access interface as exposed by BlobSeer could not be leveraged directly because it supports atomic writes and appends of contiguous regions only, which would imply the need for locking at the level of the MPI-IO layer in order to support MPI atomicity. As mentioned in Section 4, we want to avoid locking and introduce optimizations directly at the level of the storage backend. Therefore, the first step is to extend the access interface of BlobSeer such that it can describe complex non-contiguous data access in a single call.

In order to closely match the List I/O interface proposal introduced in [19], we introduce a series of versioning-oriented primitives that facilitate non-contiguous manipulations of data at the level of BLOBs.

id = CREATE(size)

This primitive creates a new BLOB and associates to it an zero-filled snapshot whose version number is 0 and is *size* bytes long. The BLOB will be identified by its *id* (the returned value). The *id* is guaranteed to be globally unique.

vw = NONCONT_WRITE(id, buffers[], offsets[], sizes[])

A NONCONT_WRITE initiates the process of generating a new snapshot of the BLOB (identified by *id*) by submitting a list of memory buffers (pointed at in buffers[]) to be overlapped over the non-contiguous regions defined by the lists offsets[] and sizes[].

The NONCONT_WRITE does not know in advance which snapshot version it will generate, as the updates are totally ordered and internally managed by the storage system. However, after the primitive returns, the caller learns about its assigned snapshot version by consulting the returned value vw. The update will eventually be applied to the snapshot $vw - 1$, thus effectively generating the snapshot vw. This snapshot version is said to be *published* when it becomes available to the readers. Note that the primitive may return before snapshot version vw is published. The publication time is unknown, but it is guaranteed that the generated snapshot will obey MPI atomicity.

NONCONT_READ(id, v, buffers[], offsets[], sizes[])

A NONCONT_READ results in replacing the contents of the memory buffers specified in the list buffers[] with the contents of the non-contiguous regions defined by the lists offsets[] and sizes[] from snapshot version v of the BLOB *id*. If v has not yet been published, the read fails.

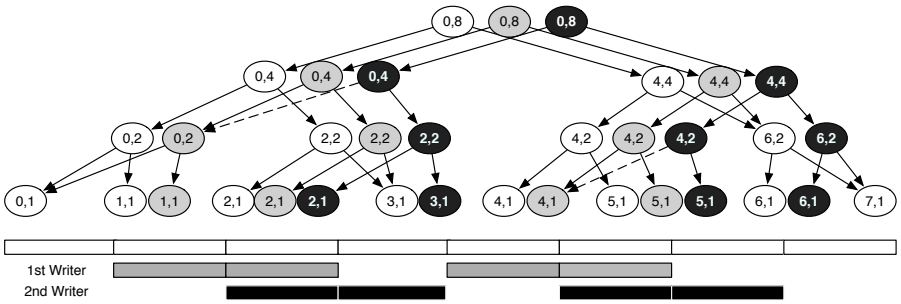


Figure 2: Metadata segement trees: whole subtrees are shared among snapshot versions

5.3 Adding support for MPI-atomicity

In Section 4, we proposed a versioning-based approach to ensure MPI atomicity efficiently under concurrency. This approach relies on the idea that non-contiguous regions can be written in parallel as a series of differences that are ordered and consolidated into independent snapshots at the level of the metadata, such that the result is equivalent to the sequential application of all differences in an arbitrary order.

The key difficulty in this context is to efficiently consolidate the differences at metadata level such that only consistent snapshots that obey MPI atomicity are published.

Since each massive BLOB snapshot is striped over a large number of storage space providers, the metadata serves the role to remember the location of each chunk in the snapshot, such that it is possible to map non-contiguous regions of the snapshot to the corresponding chunks.

5.3.1 Structure of metadata

We organize metadata as a *distributed segment tree* [20]: one such tree is associated to each snapshot of a given BLOB *id*. A segment tree is a binary tree in which each node is associated to a range of the BLOB, delimited by *offset* and *size*. We say that the node *covers* the range (*offset*, *size*). The root covers the whole BLOB snapshot, while the leaves cover single chunks (i.e. keep information about the data providers that store the chunk). For each node that is not a leaf, the left child covers the first half of the range, and the right child covers the second half. The segment tree itself is distributed at fine granularity among multiple metadata providers that form a DHT (Distributed Hash Table). This is done for scalability reasons, as a centralized solution becomes a bottleneck under concurrent accesses.

In order to avoid the overhead of rebuilding the whole segment tree for each new snapshot (which consumes both space and time), entire subtrees are shared among the snapshots, as shown in Figure 2.

5.3.2 Non-contiguous writes

A NONCONT_WRITE operation first stores all chunks that make up the non-contiguous regions on the data providers, after which it builds the metadata segment tree that is associated to the snapshot in a bottom up manner: first the leaves that hold information about where the chunks were stored and then the inner nodes up towards the root. For example, a write of two non-contiguous regions delimited by (offset, size) $= (1, 2)$ and $(4, 2)$ on a BLOB whose total size is 8 leads to the grey segment tree depicted in Figure 2.

5.3.3 Non-contiguous reads

A NONCONT_READ operation first obtains the root of the segement tree that corresponds to the snapshot version from which it needs to read, after which it descends in the segment tree towards the leaves that hold information about the chunks that make up the non-contiguous regions. Once these chunks have been established, they are fetched from the data providers and stored in the memory buffers supplied as an argument.

5.3.4 Guaranteeing MPI atomicity efficiently under concurrency

Since an already published snapshot is never modified, readers never have to synchronize with writers. The case of concurrent writers is more delicate and needs closer consideration. Concurrent writers can independently write their non-contiguous regions without any need to synchronize, because the non-contiguous regions are ordered and consolidated into consistent snapshots at metadata level. In order to perform this efficiently, we propose three important optimizations:

Minimize ordering overhead Since any arbitrary ordering that obeys MPI atomicity is valid, our goal is to minimize the time taken to establish the order in which to apply the overlappings. To this end, writers ask for a snapshot version to be assigned only after they have successfully finished writing the chunks and need to build their corresponding segment trees. The assignment process, which is the responsability of the version manager, is very simple and leads to a minimal overhead: versions are assigned on a first-come, first-served basis and practically involve only the atomic incrementation of an internal variable on the version manager. For example, in Figure 2, two concurrent writers, black and grey, wrote their non-contiguous regions. Assuming grey finished first, it was assigned version 1, while black was assigned version 2. If black finished first, the order of version assignment would have been the reverse.

Avoid synchronization for concurrent segment tree generation Once the writer obtained a new snapshot version number from the version manager, it needs to build the metadata segment tree such that it is "weaved" in a consistent fashion with the segment trees of the previous versions (i.e. the correct links to the nodes of the segment trees of the previous versions are established). We call such nodes that are linked against by the segment trees of higher versions *border nodes*. For example, the border nodes of 1st writer (grey) in Figure 2 are all belonging to the initial version (white). Under concurrency, it can happen that the border nodes of a snapshot version v belong to the segment tree of a lower

version that is in the process of being generated itself by a concurrent writer and therefore do not exist yet. For example, the border nodes of the 2nd writer (black) depend on grey nodes that have not been necesarily generated yet.

In order to avoid waiting for such border nodes to get generated, we maintain the list of all concurrent writers on the version manager and feed it as a hint to writers when they request a new snapshot version. Using this information, writers can predict what border nodes will be eventually written by the other concurrent writers and can build virtual links (which we call *metadata forward references*) to them without caring whether they exist or not, under the assumption that they will be eventually written and the segment tree will become complete. In our example, when black is assigned version number 2 it receives the hint about the existance of grey and the non-contiguous regions that grey intends to write. Using this information, black can establish the metadata forward references (dotted pattern) without waiting for grey to finish building the segment tree. When both grey and black finish, the segment trees are in a consistent state.

Lazy evaluation during border node calculation The scheme presented above greatly improves the scalability of concurrent segment tree generation, as it enables clients to independently calculate the border nodes in isolation, without the need for synchronization.

However, in addition to scalability, we aim at high performance too. To this end, we optimized the border node calculation on the client-side by introducing a lazy evaluation scheme. More precisely, we avoid precalulating the border nodes for each contiguous region individually and rather delay their evaluation until the moment when the new tree nodes are generated themselves and the links to their children need to be established. This is particularly important in the context of non-contiguous accesses, because the union of all border nodes taken from each region individually is much larger that the set of border nodes that is effectively needed.

For example, in Figure 2 grey writes two non-contigous regions. If each region is taken individually, the white node that covers $(0, 4)$ is a border node for the region delimited by (offset, size) $= (4, 2)$. Similarily, the white node that covers $(4, 8)$ is a border node for the region delimited by (offset, size) $= (1, 2)$. However, neither of them are border nodes in the end result, because their grey counterparts will eventually become the children of the grey root $(0, 8)$.

5.4 Leveraging our versioning-oriented interface at the level of the MPI-IO layer

Having obtained a storage backend implementation that directly optimizes for MPI atomicity, the next step is to efficiently leverage this storage backend at the level of the MPI-IO layer. To this end, we used ROMIO [5], a library that is part of popular MPICH2 [21] implementation.

The motivation behind this choice is the fact that ROMIO is designed in a modular fashion, making it easy to plug-in new storage backends. Architecturally, ROMIO is broken up into three layers:

- a layer that implements the MPI I/O routines in terms of an abstract I/O device that exposes a generic set of I/O primitives, called *Abstract Device interface for parallel I/O (ADIO)*;
- a layer that implements common MPI-IO optimizations that are independent of the storage backend (such as buffering and collective I/O);
- a layer that partially implements the ADIO device and needs to be extended for each storage backend explicitly.

These layers provide a complete support for MPI-IO at application level. The separation between storage backend dependent and independent code enabled us to build a lightweight ADIO module that maps the interface required by the first layer almost directly on top of the versioning-oriented access interface we introduced. More precisely, an ADIO implementation needs to provide support for both contiguous and non-contiguous writes under MPI atomicity guarantees. Our interface proposal is generic enough to handle both scenarios efficiently using a single primitive call.

6 Experimental evaluation

6.1 Overview

We conducted three series of experiments:

- An evaluation of the scalability of our approach when the same amount of data needs to be read/written into an increasing number of non-contiguous regions by the same client;
- An evaluation of the scalability of our approach when increasing the number of clients that concurrently write non-contigous regions in the same file. In this scenario, we considered the extreme case where each of the clients writes a large set of non-contiguous regions that are intentionally selected in such way as to generate a large number of overlappings that need to obey MPI atomicity;
- An evaluation of the performance of our approach using a standard benchmark, *MPI-tile-IO*, that closely simulates the access patterns of real scientific applications that split the input data into overlapped subdomains that need to be concurrently written in the same file under MPI atomicity guarantees.

In both the second and the third series of experiments, we compare our approach to the loking-based approach that leverages a POSIX-compatible file system at the level of the MPI-IO layer, which is the traditional way of addressing MPI atomicity.

In order to perform this comparison, we used two standard building blocks that are available as open-source projects: (1) the Lustre parallel file system [13], version 1.6.4.1, in its role as a high-performance, POSIX-compliant file system and (2) the default ROMIO ADIO module, which is part of the standard MPICH2 release and was specifically written for POSIX-compliant file systems.

We turned off data sieving in the default ROMIO ADIO module according to the recommendations in [10], as this greatly improves the performance of Lustre for MPI-IO. Without data sieving enabled, the ADIO module is able to take advantage of standard POSIX byte-range file locking to lock the smallest contiguous region in the file that covers all non-contigous regions that need to be read/written. Once this is done, the non-contiguous regions are read/written using a dedicated read/write call for each region individually, after which the lock is released.

6.2 Platform description

We performed our experiments on the Grid'5000 [22] testbed, a reconfigurable, controllable and monitor-able experimental Grid platform gathering 9 sites geographically distributed in France. For these experiments we used the nodes of the Rennes cluster, which are outfitted with x86_64CPUs and 4 GB of RAM. All nodes are equipped with Gigabit Ethernet cards (measured throughput: 117.5MB/s for TCP sockets with MTU = 1500 B, with a latency of 0.1 ms). We invested a significant effort in preparing the experimental setup, by implementing an automated deployment process both for Lustre and BlobSeer.

Our experiments were performed on up to 80 nodes of the Rennes cluster in the following fashion: Lustre (respectively BlobSeer) is deployed on 44 nodes, while the remaning 36 nodes are reserved to deploy a MPI ring where the MPI processes are running (each on a dedicated node).

Lustre was deployed in the following configuration: one metadata server and 43 object storage servers, each on a dedicated machine. For BlobSeer we used the following deployment setup: one version manager and one provider manager deployed on dedicated machines, while the rest of 42 nodes was used to co-deploy the data and metadata providers in pairs, one pair on each node.

6.3 Increasing number of non-contiguous regions

In the first series of experiments we evaluate the scalability of our approach when the same amount of data needs to be read/written from/into an increasing the number of non-contiguous regions by the same client.

To this end, we fix the amount of data that is read/written by the client (using NONCONT_READ and NONCONT_WRITE respectively) at 1 GB. At each step, we double the amount of non-contiguous regions into which this data is split and measure the throughput as observed on the client side. We start with a single contiguous region and end up with 1024 non-contiguous regions.

The results are shown in Figure 3. As can be observed, both reads and writes achieve a high throughput that reaches well over 80 MB/s. More importantly, the throughput drops negligibly when incresing the number of non-contiguous regions, which demonstrates excellent scalability of our approach.

6.4 Scalability under concurrency: our approach vs. locking-based

In this scenario we aim at evaluating the scalability of our approach when increasing the number of clients that concurrently write non-contiguous regions in the same file, as compared to the locking-based approach that uses Lustre.

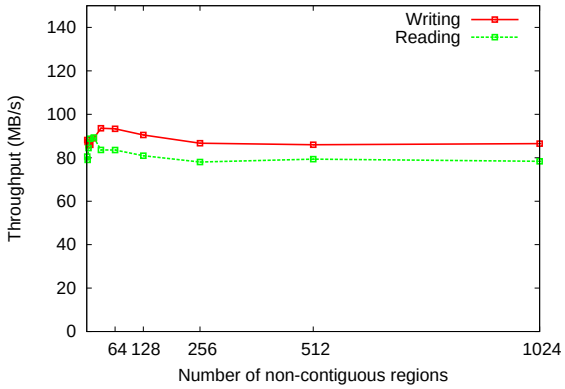


Figure 3: Scalability when the same amount of data needs to be written into an increasing number of non-contigous regions: throughput is maintained almost constant

To this end, we setup a synthetic MPI benchmark that enables us to control the MPI-IO access patterns generated by the application in such way that we generate a large number of overlappings.

More specifically, the synthetic benchmark corresponds to a special case of the overlapped subdomains depicted in Figure 1(a): namely when the subdomains (that need to be written under MPI atomicity guarantees) form a single row. Each subdomain is a matrix of 1024x1024 elements, with each element 1024 bytes large. This amounts to a total of 1 GB worth of data written by each process into 1024 non-contiguous regions, each of which is 1 MB large. Since the subdomains are arranged in a row, every MPI process (except the extremities) share a left and right overlapped subdomain with its left and respectively right neighbor. The size of the overlapping subdomain is fixed at 128x1024 elements, such that each region in the set of non-contiguous regions written by a process overlaps by 128 KB with two other regions belonging to the neighboring processes. This choice leads to a scenario that pushes both approaches to their limits: every single region generates at least one overlapping that needs to be handled in an MPI-atomic fashion.

We varied the number of MPI processes from 4 to 36 and ran the MPI benchmark both for our approach and the locking-based approach using on Lustre. Results are shown in Figure 4, where we measure the completion time to run the benchmark (i.e. the time taken by the slowest process to finish), as well as in Figure 5, where we measure the total aggregated throughput achieved by all processes (i.e. the total amount of data written by all processes divided by the completion time).

As can be observed, the completion time in the case of Lustre grows almost linearly. Since the processes are arranged in a row, the non-contiguous regions of each process are far apart, which leads to the case in which almost the whole file needs to be locked. Thus, in this extreme scenario, the accesses are practically serialized by the locking-based approach. This trend is confirmed by the total aggregated throughput as well: it remains constant at 114 MB, close to the maximal theoretical limit that can be achieved by a single client.

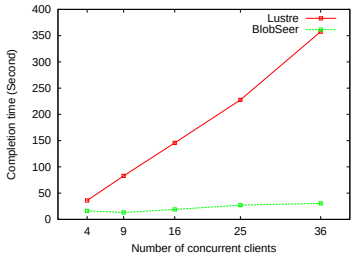


Figure 4: Our approach vs locking-based: completion time for increasing number of clients that concurrently write a large number overlapping, non-contiguous regions that are far apart (lower is better)

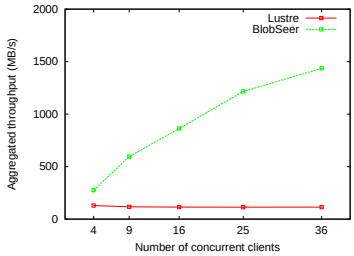


Figure 5: Our approach vs locking-based: aggregated throughput achieved by an increasing number of clients that concurrently write a large number overlapping, non-contiguous regions that are far apart (higher is better)

By contrast, the completion time in the case of our approach experiences a negligible growth, thanks to the fact that it completely avoids synchronization. This trend is confirmed by the total achieved aggregated throughput too: we can observe an increasing trend from about 300 MB/s to about 1500 MB/s for 36 concurrent clients, which more than 10 times higher that the throughput obtained by using Lustre and demonstrates excellent scalability.

6.5 MPI-tile-IO benchmark results

As a last series of experiments we evaluate the performance of our approach using a standard MPI-IO benchmarking application: *MPI-tile-IO* MPI-tile-IO closely simulates the access patterns of real scientific applications described in Section 2: overlapped subdomains that need to be concurrently written in the same file under MPI atomicity guarantees.

MPI-tile-IO sees the underlying data file as a dense two-dimensional set of subdomains (referred to in the benchmark as tiles), each of which is assigned to an MPI process, as shown in Figure 1(a). The benchmark consists in measuring the total aggregated throughput (total data written by all processes divided by the total time to complete the benchmark) achieved by all processes when they concurrently write their subdomains to the globally shared file under MPI atomicity guarantees.

This benchmark is highly configurable, enabling fine tuning of the following parameters:

- nr−tile−x: number of subdomains in the X dimension
- nr−tiles−y: number of subdomains in the Y dimension
- sz−tile−x: number of elements in the X dimension of each subdomain
- sz−tile−y: number of elements in the Y dimension of each subdomain
- sz−element: size of an element in bytes

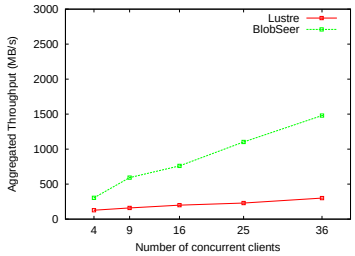


Figure 6: 1024×1024×1024 tile size

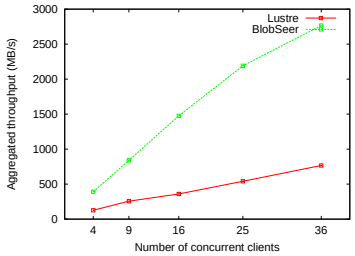


Figure 7: 32×32×1MB tile size

- overlap−x: number of elements shared between adjacent subdomains in the X dimension
- overlap−y: number of elements shared between adjacent subdomains in the Y dimension

We fixed the parameters of the MPI-tile-IO benchmark in such way as to resemble the layout of real applications as closely as possbile. More specifically, each subdomain is a 1024x1024 matrix of elements that are 1024 bytes large (i.e. sz−tile−x = sz−tile−y = sz−element = 1024). Unlike the previous series of experiments, where the processes are arranged in a row to obtain an extreme case that pushes both approaches to their limits, this time we use a realistic layout where the processes are arranged in such way that they form a square (i.e. nr−tile−x = nr−tile−y = $2 \ldots 6$, which corresponds to $4 \ldots 36$ processes). The size of the overlappings was fixed at 128x128 (i.e. overlap−x = overlap−y = 128). Thus, each process writes 1 GB worth of data, out of which 128 MB is overlapping with each of its neighbors.

We ran the MPI-tile-IO benchmark for both our approach and the locking-based approach using Lustre. The experiment was repeated 5 times and the results were averaged. Figure 6 depicts the total aggregated throughput obtained for both approaches.

As can be observed, both approaches scale. Unlike the experiment presented in Section 6.4, the choice of arranging the subdomains in such way that they form a square leads to a situation where the non-contiguous regions are closer to each other, which in turn means that a more compact contigous region needs to be locked by the Lustre-based approach. Under concurrency, this leads to a situation where different contiguous regions of the files can be locked simultaneously, thus enhancing the degree of parallelism for the locking-based approach. Nevertheless, even under such circumstances a significant amount of accesses need to be serialized, which in turn enables our approach to outperform the locking-based approach by almost 6 times for 36 concurrent processes.

To show that our approach works well even when the size of the individual regions in the non-contiguous set is large (which is a case that favours the locking-based approach), we perform another set of experiments with a different set of parameters for the MPI-tile-IO benchmark: we step up the element size from 1 KB to 1 MB, while shrinking the size of the subdomain from 1024x1024 to 32x32. This effectively keeps the total size of each subdomain the same as in the previous set of experiemnts (fixed at 1 GB), which enables a direct comparison between this scenario and the previous scenario.

Increasing the size of the elements proves to be advantageous for both approaches. In the case of our approach it leads to a coarser granularity for the data striping, which is the result of using larger chunk sizes. This in turn decreases the metadata overhead, which enables achieving a higher aggregated throughput. In the case of the locking-based approach it decreases the number of contigous writes that need to be performed for each process while increasing the size of each such write. This in turn enables achieving a higher aggregated throughput as well. However, even under these circumstances, as shown in Figure 7, our approach still outperforms the Lustre-based approach by more than 3.5 times.

7 Conclusions

In this paper, we proposed an original versioning-based mechanism that can be leveraged to efficiently address the I/O needs of data-intensive MPI applications involving data-partitioning schemes that exhibit overlapping non-contiguous I/O where MPI-IO atomicity needs to be guaranteed under concurrency.

Unlike traditional approaches that leverage POSIX-compliant parallel file systems as storage backends and employ locking schemes at the level of the MPI-IO layer, we propose to use versioning techniques as a key principle to achieve high throughputs under concurrency while guaranteeing MPI atomicity. We implemented this idea in practice by extending BlobSeer, an existing versioning-oriented, distributed data sharing service, with a non-contiguous data access interface that we directly integrated with ROMIO, a standard MPI-IO implementation. We compared our BlobSeer-based implementation with a standard locking-based approach where we used Lustre as the underlying storage backend. Our approach demonstrated excellent scalability under concurrency when compared to the Lustre-based approach. It achieved an aggregated throughput ranging from 3.5 times to 10 times higher in several experimental setups, including highly standardized MPI benchmarks specifically designed to measure the performance of MPI-IO for non-contiguous overlapped writes that need to obey MPI-atomicity semantics.

Such promising results encouraged us to purse interesting future work directions. In particular, one advantage of BlobSeer that we did not develop in this paper is to expose its versioning interface directly at application level. The ability to make use of versioning at application level brings several potential benefits, such as the case of producer-consumer workloads where for example the output of simulations is concurrently used as the input of visualizations. Using versioning at application level could avoid expensive synchronization schemes, which is an acknowledged problem of current approaches.

Acknowledgments

The experiments presented in this paper were carried out using the Grid'5000/ALADDIN-G5K experimental testbed, an initiative from the French Ministry of Research through the ACI GRID incentive action, INRIA, CNRS and RENATER and other contributing partners (see http://www.grid5000.fr/ for details).

References

[1] E. Smirni, R. Aydt, A. Chien, and D. Reed, "I/O requirements of scientific applications: An evolutionary view," in *Proceedings of 5th IEEE International Symposium on High Performance Distributed Computing*. IEEE, 2002, pp. 49–59.

[2] E. Smirni and D. A. Reed, "Lessons from characterizing the input/output behavior of parallel scientific applications," *Performance Evaluation*, vol. 33, no. 1, pp. 27–44, 1998.

[3] P. E. Crandall, R. A. Aydt, A. A. Chien, and D. A. Reed, "Input/output characteristics of scalable parallel applications," in *Proceedings of the 1995 ACM/IEEE conference on Supercomputing (CDROM)*, ser. Supercomputing '95. New York, NY, USA: ACM, 1995.

[4] N. Nieuwejaar, D. Kotz, A. Purakayastha, C. Sclatter Ellis, and M. Best, "File-access characteristics of parallel scientific workloads," *IEEE Transactions on Parallel and Distributed Systems*, vol. 7, no. 10, pp. 1075–1089, Oct. 1996.

[5] R. Thakur, W. Gropp, and E. Lusk, "On implementing mpi-i/o portably and with high performance," in *Proceedings of the sixth workshop on I/O in parallel and distributed systems*, ser. IOPADS '99. New York, NY, USA: ACM, 1999, pp. 23–32.

[6] *Information technology - Portable Operating System Interface (POSIX) Operating System Interface (POSIX)*. Institute of Electrical & Electronics Engineers, 2009.

[7] I. F. Haddad, "Pvfs: A parallel virtual file system for linux clusters," *Linux J.*, vol. 2000, November 2000.

[8] R. Ross, R. Latham, W. Gropp, R. Thakur, and B. Toonen, "Implementing mpi-io atomic mode without file system support," in *Proceedings of the Fifth IEEE International Symposium on Cluster Computing and the Grid (CCGrid'05)*, ser. CCGRID '05, vol. 2. Washington, DC, USA: IEEE Computer Society, 2005, pp. 1135–1142.

[9] S. Sehrish, J. Wang, and R. Thakur, "Conflict detection algorithm to minimize locking for mpi-io atomicity," in *Proceedings of the 16th European PVM/MPI Users' Group Meeting on Recent Advances in Parallel Virtual Machine and Message Passing Interface*. Berlin, Heidelberg: Springer-Verlag, 2009, pp. 143–153.

[10] A. Ching, W.-k. Liao, A. Choudhary, R. Ross, and L. Ward, "Noncontiguous locking techniques for parallel file systems," in *Proceedings of the 2007 ACM/IEEE Conference on Supercomputing*, Nov. 2007, pp. 1–12.

[11] W.-K. Liao, A. Choudhary, K. Coloma, G. Thiruvathukal, L. Ward, E. Russell, and N. Pundit, "Scalable implementations of mpi atomicity for concurrent overlapping i/o," in *Parallel Processing, 2003. Proceedings. 2003 International Conference on*, Oct. 2003, pp. 239–246.

[12] F. Schmuck and R. Haskin, "Gpfs: A shared-disk file system for large computing clusters," in *Proceedings of the 1st USENIX Conference on File and Storage Technologies*, ser. FAST '02. Berkeley, CA, USA: USENIX Association, 2002.

[13] P. Schwan, "Lustre: Building a file system for 1000-node clusters," in *Proceedings of the Linux Symposium*, 2003.

[14] G. Bryan, "http://www.mmm.ucar.edu/people/bryan/cm1/."

[15] O. Rodeh, "B-trees, shadowing, and clones," *Trans. Storage*, vol. 3, no. 4, pp. 1–27, 2008.

[16] B. Nicolae, G. Antoniu, L. Bougé, D. Moise, and A. Carpen-Amarie, "Blobseer: Next generation data management for large scale infrastructures," *Journal of Parallel and Distributed Computing*, 2010, in press.

[17] 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*, pp. 404–416, 2009.

[18] B. Nicolae, D. Moise, G. Antoniu, L. Bougé, and M. Dorier, "BlobSeer: Bringing high throughput under heavy concurrency to Hadoop Map-Reduce applications," in *IEEE International Symposium on Parallel & Distributed Processing (IPDPS)*. IEEE, 2010, pp. 1–11.

[19] A. Ching, A. Choudhary, W.-k. Liao, R. Ross, and W. Gropp, "Noncontiguous i/o through pvfs," in *Proceedings of the IEEE International Conference on Cluster Computing*, ser. CLUSTER '02. Washington, DC, USA: IEEE Computer Society, 2002, pp. 405–.

[20] C. Zheng, G. Shen, S. Li, and S. Shenker, "Distributed segment tree: Support of range query and cover query over dht," in *The 5th International Workshop on Peer-to-Peer Systems (IPTPS)*, 2006.

[21] W. Gropp, E. Lusk, and A. Skjellum, *Using MPI: Portable Parallel Programming with the Message Passing Interface*. MIT Press, 1994.

[22] Y. Jégou, S. Lantéri, J. Leduc, M. Noredine, G. Mornet, R. Namyst, P. Primet, B. Quetier, O. Richard, E.-G. Talbi, and T. Iréa, "Grid'5000: a large scale and highly reconfigurable experimental grid testbed," *International Journal of High Performance Computing Applications*, vol. 20, no. 4, pp. 481–494, November 2006.

Centre de recherche INRIA Rennes – Bretagne Atlantique
IRISA, Campus universitaire de Beaulieu - 35042 Rennes Cedex (France)

Centre de recherche INRIA Bordeaux – Sud Ouest : Domaine Universitaire - 351, cours de la Libération - 33405 Talence Cedex
Centre de recherche INRIA Grenoble – Rhône-Alpes : 655, avenue de l'Europe - 38334 Montbonnot Saint-Ismier
Centre de recherche INRIA Lille – Nord Europe : Parc Scientifique de la Haute Borne - 40, avenue Halley - 59650 Villeneuve d'Ascq
Centre de recherche INRIA Nancy – Grand Est : LORIA, Technopôle de Nancy-Brabois - Campus scientifique
615, rue du Jardin Botanique - BP 101 - 54602 Villers-lès-Nancy Cedex
Centre de recherche INRIA Paris – Rocquencourt : Domaine de Voluceau - Rocquencourt - BP 105 - 78153 Le Chesnay Cedex
Centre de recherche INRIA Saclay – Île-de-France : Parc Orsay Université - ZAC des Vignes : 4, rue Jacques Monod - 91893 Orsay Cedex
Centre de recherche INRIA Sophia Antipolis – Méditerranée : 2004, route des Lucioles - BP 93 - 06902 Sophia Antipolis Cedex

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