

BlobSeer: Bringing High Throughput under Heavy Concurrency to Hadoop Map/Reduce Applications

Bogdan Nicolae, Diana Moise, Gabriel Antoniu, Luc Bougé, Matthieu Dorier

▶ **To cite this version:**

Bogdan Nicolae, Diana Moise, Gabriel Antoniu, Luc Bougé, Matthieu Dorier. BlobSeer: Bringing High Throughput under Heavy Concurrency to Hadoop Map/Reduce Applications. [Research Report] RR-7140, INRIA. 2009, pp.20. inria-00440312

HAL Id: inria-00440312
https://inria.hal.science/inria-00440312

Submitted on 10 Dec 2009

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

BlobSeer: Bringing High Throughput under Heavy Concurrency to Hadoop Map/Reduce Applications

Bogdan Nicolae — Diana Moise — Gabriel Antoniu — Luc Bougé — Matthieu Dorier

N° 7140

December 2009



ISSN 0249-6399 ISRN INRIA/RR--7140--FR+ENG

INSTITUT NATIONAL
DE RECHERCHE
EN INFORMATIQUE
ET EN AUTOMATIQUE

INRIA

centre de recherche
RENNES - BRETAGNE ATLANTIQUE

BlobSeer: Bringing High Throughput under Heavy Concurrency to Hadoop Map/Reduce Applications

Bogdan Nicolae*, Diana Moise†, Gabriel Antoniu†, Luc Bougé‡, Matthieu Dorier‡

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

Rapport de recherche n° 7140 — December 2009 — 20 pages

Abstract: Hadoop is a software framework supporting the Map/Reduce programming model. It relies on the Hadoop Distributed File System (HDFS) as its primary storage system. The efficiency of HDFS is crucial for the performance of Map/Reduce applications. We substitute the original HDFS layer of Hadoop with a new, concurrency-optimized data storage layer based on the BlobSeer data management service. Thereby, the efficiency of Hadoop is significantly improved for data-intensive Map/Reduce applications, which naturally exhibit a high degree of data access concurrency. Moreover, BlobSeer's features (built-in versioning, its support for concurrent append operations) open the possibility for Hadoop to further extend its functionalities. We report on extensive experiments conducted on the Grid'5000 testbed. The results illustrate the benefits of our approach over the original HDFS-based implementation of Hadoop.

Key-words: Distributed file systems; High-performance systems; High throughput; Large-scale; Heavy access concurrency; Map/Reduce applications; Hadoop; BlobSeer.

Contact author: Gabriel.Antoniu@inria.fr.

A slightly revised version of this work will be published in the Proceedings of the 24th IEEE International Parallel and Distributed Processing Symposium (IPDPS 2010), Atlanta, April 19-23, 2010

* University of Rennes 1/IRISA, Rennes, France
† INRIA/IRISA, Rennes, France
‡ ENS Cachan Bretagne/IRISA, Rennes, France

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

BlobSeer: augmenter la bande passante dans un contexte de forte concurrence pour les applications Map/Reduce sur Hadoop

Résumé : Hadoop est un environnement logiciel pour la mise en œuvre du modèle de programmation Map/Reduce. Il s'appuie principalement sur le système de gestion de fichiers distribué HDFS. L'efficacité de HDFS est un paramètre crucial de la performance des applications Map/Reduce. Nous proposons de remplacer la couche HDFS de Hadoop par une nouvelle couche de stockage des données qui soit optimisée pour une utilisation concurrente. Cette nouvelle couche s'appuie sur le service de gestion de données BlobSeer. Nous montrons que l'efficacité de Hadoop est ainsi améliorée de manière significative pour des applications Map/Reduce qui manipulent intensivement les données: en effet, elles offrent naturellement un haut degré de concurrence. De plus, les fonctionnalités spécifiques de BlobSeer (gestion intégrée des versions, support pour les opérations "append" concurrentes) permettent d'envisager d'étendre les fonctionnalités de Hadoop. Nous rendons compte d'une campagne intensive d'expériences menée sur l'instrument Grid'5000. Les résultats illustrent les bénéfices de notre approche par rapport à l'implémentation primitive de Hadoop fondée sur HDFS.

Mots-clés : Système de gestion de fichiers distribué; système haute performance; grande bande passante; grande échelle; accès hautement concurrents; applications Map/Reduce; Hadoop; BlobSeer.

Contents

1 Introduction **3**

2 Specialized file systems for data-intensive Map/Reduce applications **4**
2.1 Requirements for the storage layer 4
2.2 Dedicated file systems for Map/Reduce 4

3 BlobSeer as a concurrency-optimized file system for Hadoop **6**
3.1 Design overview of BlobSeer 6
3.2 Integrating BlobSeer with Hadoop 9
3.3 BlobSeer: detailed architecture 10
3.4 Zooming on reads 11
3.5 Zooming on writes 11

4 Experimental evaluation **12**
4.1 Microbenchmarks 13
4.2 Scenario 1: single writer, single file 14
4.3 Scenario 2: concurrent reads, shared file 15
4.4 Scenario 3: Concurrent appends, shared file 16
4.5 Higher-level experiments with Map/Reduce applications 17

5 Conclusion **18**

1 Introduction

Map/Reduce [5] is a parallel programming paradigm successfully used by large Internet service providers to perform computations on massive amounts of data. After being strongly promoted by Google, it has also been implemented by the open source community through the Hadoop [7] project, maintained by the Apache Foundation and supported by Yahoo! and even by Google itself. This model is currently getting more and more popular as a solution for rapid implementation of distributed data-intensive applications.

At the core of the Map/Reduce frameworks stays a key component: the storage layer. To enable massively parallel data processing to a high degree over a large number of nodes, the storage layer must meet a series of specific requirements (discussed in Section 2), that are not part of design specifications of traditional distributed file systems employed in the HPC communities: these file systems typically aim at conforming to well-established standards such as POSIX and MPI-IO. To address these requirements, specialized file systems have been designed, such as HDFS [8], the default storage layer of Hadoop. HDFS has however some difficulties to sustain a high throughput in the case of concurrent accesses to the same file. Moreover, many desirable features are missing altogether, such as the support for versioning and for concurrent updates to the same file.

We substitute the original data storage layer of Hadoop with a new, concurrency-optimized storage layer based on BlobSeer, a data management service we developed with the goal of supporting efficient, fine-grain access to massive, distributed data accessed under heavy concurrency. By using BlobSeer

instead of its default storage layer, Hadoop significantly improves its sustained throughput in scenarios that exhibit highly concurrent accesses to shared files. We report on extensive experimentation both with synthetic microbenchmarks and real Map/Reduce applications. The results illustrate the benefits of our approach over the original HDFS-based implementation of Hadoop. Moreover we support additional features such as efficient concurrent appends, concurrent writes at random offsets and versioning. These features could be leveraged to extend or improve functionalities in future versions of Hadoop or other Map/Reduce frameworks.

2 Specialized file systems for data-intensive Map/Reduce applications

2.1 Requirements for the storage layer

Map/Reduce applications typically crunch ever growing data sets of billions of small records. Storing billions of KB-sized records in separate tiny files is both unfeasible and hard to handle, even if the storage layer would support it. For this reason, data sets are usually packed together in *huge files* whose size reaches the order of several hundreds of GB.

The key strength of the Map/Reduce model is its inherently high parallelization of the computation, that enables processing of PB of data in a couple of hours on large clusters consisting of several thousand nodes. This has several consequences for the storage backend. Firstly, since data is stored in huge files, the computation will have to process small parts of these huge files concurrently. Thus, the storage layer is expected to provide efficient *fine-grain access* to the files. Secondly, the storage layer must be able to sustain a *high throughput* in spite of *heavy access concurrency* to the same file, as thousands of clients simultaneously access data.

Dealing with of huge amounts of data is difficult in terms of manageability. Simple mistakes that may lead to loss of data can have disastrous consequences since gathering such amounts of data requires considerable effort investment. *Versioning* in this context becomes an important feature that is expected from the storage layer. Not only it enables rolling back undesired changes, but also branching a dataset into two independent datasets that can evolve independently. Obviously, versioning should have a minimal impact both on performance and on storage space overhead.

Finally, another important requirement for the storage layer is its ability to expose an interface that enables the application to be *data-location aware*. This allows the scheduler to use this information to place computation tasks close to the data. This reduces network traffic, contributing to a better global data throughput.

2.2 Dedicated file systems for Map/Reduce

These critical needs of data-intensive distributed applications have not been addressed by classical, POSIX-compliant distributed file systems. Therefore, Google introduced GoogleFS [6] as a storage backend that provides the right

abstraction for their Map/Reduce data processing framework. Then, other specialized file systems emerged: companies such as Yahoo! and Kosmix followed this trend by emulating the GoogleFS architecture with the Hadoop Distributed File System (HDFS, [8]) and CloudStore [4].

Essentially, GoogleFS splits files into fixed-sized 64 MB chunks that are distributed among chunkservers. Both metadata that describes the directory structure of the file system, and metadata that describes the chunk layout are stored on a centralized master server. Clients that need to access a file first contact this server to obtain the location of the chunks that correspond to the range of the file they are interested in. Then, they directly interact with the corresponding chunkservers. GoogleFS is optimized to sustain a high throughput for concurrent reads/appends from/to a single file, by relaxing the semantic consistency requirements. It also implements support for cheap snapshooting and branching.

Hadoop Map/Reduce is a framework designed for easily writing and efficiently processing Map/Reduce applications. The framework consists of a single master *jobtracker*, and multiple slave *tasktrackers*, one per node. The *jobtracker* is responsible for scheduling the jobs' component tasks on the slaves, monitoring them and re-executing the failed tasks. The *tasktrackers* execute the tasks as directed by the master. HDFS is the default storage backend that ships with the Hadoop framework. It was inspired by the architecture of GoogleFS. Files are also split in 64 MB blocks that are distributed among *datanodes*. A centralized *namenode* is responsible to maintain both chunk layout and directory structure metadata. Read and write requests are performed by direct interaction with the corresponding *datanodes* and do not go through the *namenode*.

In Hadoop, reads essentially work the same way as with GoogleFS. However, HDFS has a different semantics for concurrent write access: it allows only one writer at a time, and, once written, data cannot be altered, neither by overwriting nor by appending. Several optimization techniques are used to significantly improve data throughput. First, HDFS employs a client side buffering mechanism for small read/write accesses. It prefetches data on reading. On writing, it postpones committing data after the buffer has reached at least a full chunk size. Actually, such fine-grain accesses are dominant in Map/Reduce applications, which usually manipulate small records. Second, Hadoop's job scheduler (the *jobtracker*) places computations as close as possible to the data. For this purpose, HDFS explicitely exposes the mapping of chunks over *datanodes* to the Hadoop framework.

With cloud computing becoming more and more popular, providers such as Amazon started offering Map/Reduce platforms as a service. Amazon's initiative, Elastic MapReduce [2], employs Hadoop on their Elastic Compute Cloud infrastructure (EC2, [1]). The storage backend used by Hadoop is Amazon's Simple Storage Service (S3, [3]). The S3 framework was designed with simplicity in mind, to handle objects that may reach sizes in the order of GB: the user can write, read, and delete objects simply identified by an unique key. The access interface is based on well-established standards such as SOAP. Careful consideration was invested into using decentralized techniques and designing operations in such way as to minimize the need for concurrency control. A fault tolerant layer enables operations to continue with minimal interruption. This allows S3 to be highly scalable. On the downside however, simplicity comes at a cost: S3 provides limited support for concurrent accesses to a single object.

Other efforts aim at adapting general-purpose distributed file systems from the HPC community to the needs of the Map/Reduce applications. For instance, PVFS (Parallel Virtual File System) and GPFS (General Parallel File System, from IBM) have been adapted to serve as a storage layer for Hadoop. GPFS [13] is part of the *shared-disk file systems* class, that use a pool of block-level storage, shared and distributed across all the nodes in the cluster. The shared storage can be directly accessed by clients, with no interaction with an intermediate server. Integrating GPFS with the Hadoop framework, involves overcoming some limitations: GPFS supports a maximal block size of 16 MB, whereas Hadoop often makes use of data in 64 MB chunks; Hadoop's *jobtracker* must be aware of the block location, while GPFS (like all parallel file systems) exposes a POSIX interface. PVFS [12] belongs to a second class of parallel file systems, *object-based file systems* which separate the nodes that store the data from the ones that store the medatata (file information, and file block location). When a client wants to access a file, it must first contact the metadata server and then directly access the data on the data servers indicated by the metadata server. In [14], it is described the way PVFS was integrated with Hadoop, by adding a layer on top of PVFS. This layer enhanced PVFS with some features that HDFS already provides to the Hadoop framework: performing read-ahead buffering, exposing the data layout and emulating replication.

The above work has been a source of inspiration for our approach. Thanks to the specific features of BlobSeer, we could address several limitations of HDFS highlighted in it.

3 BlobSeer as a concurrency-optimized file system for Hadoop

In this section we introduce *BlobSeer*, a system for managing massive data in a large-scale distributed context [10]. Its efficient version-oriented design enables lock-free access to data, and thereby favors scalablity under heavy concurrency. Thanks to its decentralized data and metadata management, it provides high data throughput [11]. The goal of this paper is to show how BlobSeer can be extended into an filesystem for *Hadoop*, and thus used as an efficient storage backend for Map/Reduce applications.

3.1 Design overview of BlobSeer

The goal of *BlobSeer* is to provide support for data-intensive distributed applications. No hypothesis whatsoever is made about the structure of the data at stake: they are viewed as huge, flat sequences of bytes, often called *BLOBs* (*Binary Large OBjects*). We especially target applications that process BLOBs in a *fine-grain* manner. This is the typical case of Map/Reduce applications, indeed: workers usually access pieces of up to 64 MB from huge input files, whose size may reach hundreds of GB.

A client of BlobSeer manipulates BLOBs by using a simple interface that allows to: create a new empty BLOB; append data to an existing BLOB; read-/write a subsequence of bytes specified by an offset and a size from/to an existing BLOB. Each BLOB is identified by a unique id in the system.

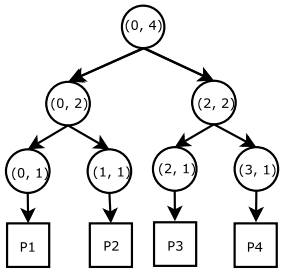


Figure 1: Metadata tree after writing the first 4 blocks of a BLOB

Versioning is built in BlobSeer at the earliest stage of design. Each time a write or append is performed on a BLOB, a new snapshot reflecting the changes is generated instead of overwriting any existing data. This new snapshot is labeled with an incremental version number, so that all past versions of the BLOB can potentially be accessed, at least as long as they have not been garbaged for the sake of storage space.

The version numbers are assigned and managed by the system. In order to read a part of the BLOB, the client must specify both the unique id of the BLOB and the snapshot version it desires to read from. A special call allows the client to find out the latest version of a particular BLOB, but the client is allowed to read any past version of the BLOB.

Although each write or append generates a new version, only the differential patch is actually stored, so that storage space is saved at far as possible. The new snapshot shares all unmodified data and most of the associated metadata with the previous versions, as we will see further in this section. Such an implementation further facilitates the implementation of advanced features such as rollback and branching, since data and metadata corresponding to past versions remain available in the system and can easily be accessed.

The goal of BlobSeer is to sustain *high throughput* under *heavy access concurrency* in reading, writing and appending. This is achieved thanks to the combination of various techniques, including: data striping, distributed metadata, version-based design, lock-free data access.

Data striping. BlobSeer relies on striping: each BLOB is made up of blocks of a fixed size. To optimize BlobSeer for Map/Reduce applications, we set this size to the size of the data piece a Map/Reduce worker is supposed to process (i.e., 64 MB in the experiments below with Hadoop, equal to the chunk size in HDFS). These blocks are distributed among the storage nodes. We use a load balancing strategy that aims at evenly distributing the blocks among these nodes. As described in Section 4.3, this has a major positive impact in sustaining a high throughput when many concurrent readers access different parts of the same file.

Distributed metadata. A BLOB is accessed by specifying a version number and a range of bytes delimited by an offset and a size. BlobSeer manages additional metadata to map a given range and a version to the physical nodes where the corresponding blocks are located. We organize metadata as a *distributed*

segment tree [15]: one such tree is associated to each version 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. 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. Each leaf covers a single block of the BLOB. Figure 1 illustrates such a metadata tree for a 4-block. To favor efficient concurrent access to metadata, tree nodes are distributed: they are stored on the metadata providers using a DHT (Distributed Hash Table). Each tree node is identified in the DHT by its version and by the range specified through the offset and the size it covers. Such a metadata tree is created when the first blocks of the blob are written, for the range covered by those blocks. Then, to avoid the overhead (in time and space!) of rebuilding such a tree for the subsequent updates, we create new tree nodes only for the ranges that *do* intersect with the range of the update.

Note that metadata decentralization has a significant impact on the global throughput, as demonstrated in [11]: it avoids the bottleneck created by concurrent accesses in the case of a centralized metadata server in most distributed file systems, including HDFS. A detailed description of the algorithms we use to manage metadata can be found in [10]: due to space constraints, we will not develop them further in this paper.

Version-based, lock-free, concurrency-optimized data access. BlobSeer relies on a versioning-based concurrency control algorithm that maximizes the number of operations performed in parallel in the system. It is done by avoiding synchronization as much as possible, both at the data and metadata levels. The key idea is amazingly simple: no existing data or metadata is ever modified! First, any writer or appender writes its new data blocks, by storing the differential patch. Then, in a second phase, the version number is allocated and the new metadata referring to these blocks are generated. The first phase consists in actually writing the new data on the distributed storage nodes. The concurrent writers can proceed with full parallelism, without any synchronization. In the second phase, the new metadata are then "weaved" together with the metadata of the versions with a lower number. The crucial observation is that this second phase can also be mostly concurrent. The only global synchronization requirement is that the order in which the completion of the concurrent writes occurs must respect the order in which the version numbers have been assigned. This is transparently ensured by the system, without requiring any explicit synchronization by the user. Thereby, the algorithm creates the illusion of a fully independent snapshot generation. This allows write/write concurrency at data level, while still preserving serialization and atomicity.

Since each writer or appender generates new data/metadata and never modifies existing data/metadata, readers are completely decoupled from them. Readers can thus proceed with full concurrency with respect to writers and appenders (and vice-versa), both for data and metadata access. We can thus claim that our approach supports read/read, read/write and write/write concurrency by design. This significantly overpasses the capabilities of HDFS, which only allows a single writer to proceed at a time. The experimental results presented in Section 4 clearly support our claim.

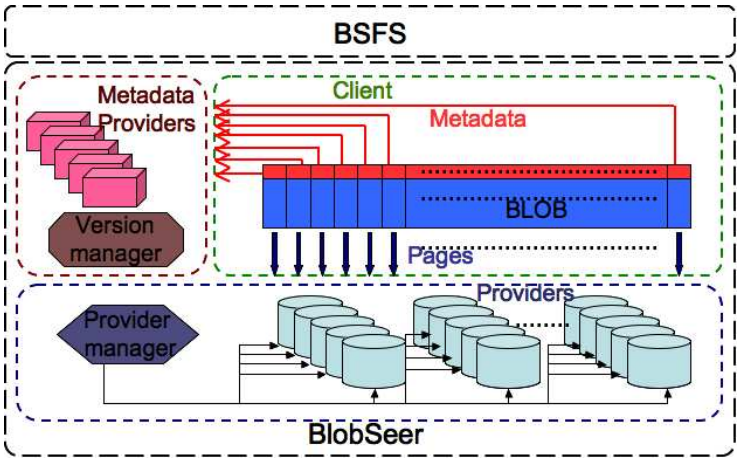


Figure 2: BlobSeer's architecture. *The BSFS layer enables Hadoop to use BlobSeer as a storage backend through a file system interface.*

3.2 Integrating BlobSeer with Hadoop

The Hadoop Map/Reduce framework accesses its default storage backend (HDFS) through a clean, specific Java API. This API exposes the basic operations of a file system: read, write, append, etc. To make Hadoop benefit from BlobSeer's properties, we implemented this API on top of BlobSeer. We call this higher layer the *BlobSeer File System* (BSFS): it enables BlobSeer to act as a storage backend file system for Hadoop. To enable a fair comparison of BSFS with HDFS, we addressed several performance-oriented issues highlighted in [14]. They are briefly discussed below.

File system namespace. The Hadoop framework expects a classical hierarchical directory structure, whereas BlobSeer provides a flat structure for BLOBs. For this purpose, we had to design and implement a specialized *namespace manager*, which is responsible for maintaining a file system namespace, and for mapping files to BLOBs. For the sake of simplicity, this entity is centralized. Careful consideration was given to minimize the interaction with this namespace manager, in order to fully benefit from the decentralized metadata management scheme of BlobSeer. Our implementation of Hadoop's file system API only interacts with it for operations like file opening and file/directory creation/deletion/renaming. Access to the actual data is performed by a direct interaction with BlobSeer through read/write/append operations on the associated BLOB, which fully benefit from BlobSeer's efficient support for concurrency.

Data prefetching. Hadoop manipulates data sequentially in small chunks of a few KB (usually, 4 KB) at a time. To optimize throughput, HDFS implements a caching mechanism that prefetches data for reads, and delays committing data for writes. Thereby, physical reads and writes are performed with data sizes large enough to compensate for network traffic overhead. We implemented a similar caching mechanism in BSFS. It prefetches a whole block when the

requested data is not already cached, and delays committing writes until a whole block has been filled in the cache.

Affinity scheduling: exposing data distribution. In a typical Hadoop deployment, the same physical nodes act both as storage elements and as computation workers. Therefore, the Hadoop scheduler strives at placing the computation as close as possible to the data: this has a major impact on the global data throughput, given the huge volume of data being processed. To enable this scheduling policy, Hadoop's file system API exposes a call that allows Hadoop to learn how the requested data is split into blocks, and where those blocks are stored. We address this point by extending BlobSeer with a new primitive. Given a specified BLOB id, version, offset and size, it returns the list of blocks that make up the requested range, and the addresses of the physical nodes that store those blocks. Then, we simply map Hadoop's corresponding file system call to this primitive provided by BlobSeer.

3.3 BlobSeer: detailed architecture

BlobSeer consists of a series of distributed communicating processes. Figure 2 illustrates the processes and their interactions between them.

Clients create, read, write and append data from/to BLOBs. Clients can access the BLOBs with full concurrency, even if they all access the same BLOB.

Data providers physically store the blocks generated by appends and writes. New data providers may dynamically join and leave the system. In the context of Hadoop Map/Reduce, the nodes hosting data providers typically also act as computing elements as well. This enables them to benefit from the scheduling strategy of Hadoop, which aims at placing the computation as close as possible to the data.

The provider manager keeps information about the available storage space and schedules the placement of newly generated blocks. For each such block to be stored, it selects the data providers according to a load balancing strategy that aims at evenly distributing the blocks across data providers.

Metadata providers physically store the metadata that allows identifying the blocks that make up a snapshot version. We use a distributed metadata management scheme to enhance concurrent access to metadata. The nodes hosting metadata providers may act as computing elements as well.

The version manager is in charge of assigning snapshot version numbers in such a way that serialization and atomicity of writes and appends is guaranteed. It is typically hosted on a dedicated node.

The namespace manager is not part of the BlobSeer. It is an additional entity introduced for BSFS, the higher-level file system layer. It maintains a file system namespace, and maps files in the namespace to BLOBs. It is typically hosted on a dedicated node.

3.4 Zooming on reads

To read data, the client first needs to find out the BLOB corresponding to the requested file. This information is typicaly available locally (as it has typically been requested from the namespace manager when the file was opened). Then the client must specify the version number it desires to read from, as well as the offset and size of the range to be read. The client may also call a special primitive first, to find out the latest version available in the system at the time this primitive was invoked. In practice, since Hadoop's file system API does not support versioning yet, this call is always issued in the current implementation.

Next, the read operation in BSFS follows BlobSeer's sequence of steps for reading a range within a BLOB. The corresponding distributed algorithm, describing the interactions between the client, the version manager, the distributed data and metadata providers are presented and discussed in detail in [10]. The main global steps can be summarized as follows. The client queries the version manager about the requested version of the BLOB. The version manager forwards the query to the metadata providers, which send to the client the metadata that corresponds to the blocks that make up the requested range. When the location of all these blocks was determined, the client fetches the blocks from the data providers. These requests are sent asynchronously and processed in parallel by the data providers. Note that the first and the last block in the sequence of blocks for the requested range may not need to be fetched completely, as the requested range may be unaligned to full blocks. In this case, the client fetches only the required parts of the extremal blocks.

3.5 Zooming on writes

To write data, the client first splits the data to be written into a list of blocks that correspond to the requested range. Then, it contacts the provider manager, requesting a list of providers capable of storing the blocks: one provider for each block. Blocks are then written in parallel to the providers allocated by the provider manager. If, for some reason, writing of a block fails, then the whole write fails. Otherwise the client proceeds by contacting the version manager to announce its intent to update the BLOB. As highlighted in Section 3.1, concurrent writers of different blocks of the same file can perform this first step with full parallelism. Subsequently, the version manager assigns to each write request a new snapshot version number. This number is used by the client to generate new metadata, weave it together with *existing metadata*, and store it on the distributed metadata providers, in order to create the illusion of a new standalone snapshot.

Note that the term "existing metadata" covers two cases. First, it refers to metadata corresponding to previous, completed writes. But it also refers to metadata generated by still active concurrent writers that were assigned a lower version number (i.e., they have written the data, but they have not finished writing the metadata)! In particular, such concurrent writers might be in the process of generating and writing metadata, on which the client shall depend when weaving its own metadata. To deal with this situation, the version manager hints the client on such dependencies. In some sense, the client is able to *predict* the values corresponding to the metadata that is being written by the concurrent writers that are still in progress. It can thus proceed concurrently

with the other writers, rather than waiting for them to finish writing their metadata. The reader can refer to [10] for further details on how we handle metadata for concurrent writers.

Once metadata was successfully written to the metadata providers, the client notifies the version manager of success, and returns to the user. Observe that the version manager needs to keep track of all writers concurrently active, and delay completing a new snapshot version until all writers that were assigned a lower version number reported success. The detailed algorithm for writing is provided in [10].

The append operation is identical to the write operation, except for a single difference: the offset of the range to be appended is unknown at the time the append is issued. It is eventually fixed by the version manager at the time the version number is assigned. It is set to the size of the snapshot corresponding to the preceding version number. Again, observe that the writing of this snapshot may still be in progress.

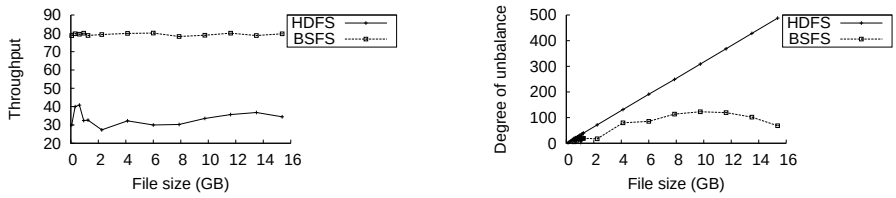
4 Experimental evaluation

Platform description. To evaluate the benefits of using BlobSeer as the storage backend for Map/Reduce applications we used Yahoo!'s release of Hadoop v.0.20.0 (which is essentially the main release of Hadoop with some minor patches designed to enable Hadoop to run on the Yahoo! production clusters). We chose this release because it is freely available and enables us to experiment with a framework that is both stable and used in production on Yahoo!'s clusters.

We performed our experiments on the Grid'5000 [9] testbed, a reconfigurable, controllable and monitorable experimental Grid platform gathering 9 sites geographically distributed in France. We used the clusters located in Sophia-Antipolis, Orsay and Lille. Each experiment was carried out within a single such cluster. The nodes are outfitted with x86_64 CPUs and 4 GB of RAM for the Rennes and Sophia clusters (2 GB for the cluster located in Orsay). Intracluster bandwidth is 1 Gbit/s (measured: 117.5 MB/s for TCP sockets with MTU = 1500 B), intracluster latency is 0.1 ms. A significant effort was invested in preparing the experimental setup, by defining an automated deployment process for the Hadoop framework both when using BlobSeer and HDFS as the storage backend. We had to overcome nontrivial node management and configuration issues to reach this point.

Overview of the experiments. In a first phase, we have implemented a set of microbenchmarks that write/read and append data to files through Hadoop's file system API and have measured the achieved throughput as more and more concurrent clients access the file system. This synthetic setup has enabled us to control the access pattern to the file system and focus on different scenarios that exhibit particular access patterns. We can thus directly compare the respective behavior of BSFS and HDFS in these particular synthetic scenarios.

In a second phase, our goal was to get a feeling of the impact of BlobSeer at the application level. We have run two standard Map/Reduce applications from the Hadoop release, both with BSFS and with HDFS. We have evaluated the impact of using BSFS instead of HDFS on the total job execution time as



(a) Performance of HDFS and BSFS when a single client writes to a single file

(b) Load-balancing evaluation

Figure 3: Single writer results

the number of available Map/Reduce workers progressively increases. Note that Hadoop Map/Reduce applications run out-of-the-box in an environment where Hadoop uses BlobSeer as a storage backend, just like in the original, unmodified environment of Hadoop. This was made possible thanks to the Java file system interface we provided with BSFS, on top of BlobSeer.

4.1 Microbenchmarks

We have first defined several scenarios aiming at evaluating the throughput achieved by BSFS and HDFS when the distributed file system is accessed by a single client or by multiple, concurrent clients, according to several specific access patterns. In this paper we have focused the following patterns, often exhibited by Map/Reduce applications:

- a single process writing a huge distributed file;
- concurrent readers reading different parts of the same huge file;
- concurrent writers appending data to the same huge file.

The aim of these experiments is of course to evaluate which benefits can be expected when using a concurrency-optimized storage service such as BlobSeer for highly-parallel Map-Reduce applications generating such access patterns. The relevance of these patterns is discussed in the following subsections, for each scenario. Additional scenarios with other different access patterns are currently under investigation.

In each scenario, we first measure the throughput achieved when a single client performs a set of operations on the file system. Then, we gradually increase the number of clients performing the same operation concurrently and measure the average throughput per client. For any fixed number N of concurrent clients, the experiment consists in two phases: we deploy of HDFS (respectively BSFS) on a given setup, then we run the test scenario.

In the deployment phase, HDFS (respectively BSFS) is deployed on 270 machines from the same cluster of Grid'5000. For HDFS, we deploy one *namenode* on a dedicated machine; the remaining nodes are used for the *datanodes* (one *datanode* per machine). On the same number of nodes, we deploy BSFS as

follows: one version manager, one provider manager, one node for the namespace manager, 20 metadata providers; the remaining nodes are used as data providers. Each entity is deployed on a a separate, dedicated machine.

For the measurement phase, a subset of N machines is chosen from the set of machines where *datanodes*/providers are running. The clients are then launched simultaneously on this subset of machines, individual throughput is collected and is then averaged. These steps are repeated 5 times for better accuracy (which is enough, as the corresponding standard deviation proved to be low).

4.2 Scenario 1: single writer, single file

We first measure the performance of HDFS/BSFS when a single client writes a file whose size gradually increases. This test consists in sequentially writing a unique file of $N \times 64$ MB, in blocks of 64 MB (N goes from 1 to 246). The size of HDFS's chunks is 64 MB, and so is the block size configured with BlobSeer in this case. The goal of this experiment is to compare the block allocation strategies that HDFS and BSFS use in distributing the data across *datanodes* (respectively data providers). The policy used by HDFS consists in writing *locally* whenever a write is initiated on a *datanode*. To enable a fair comparison, we chose to always deploy clients on nodes where no *datanode* has previously been deployed. This way, we make sure that HDFS will distribute the data among the *datanodes*, instead of locally storing the whole file. BlobSeer's default strategy consists in allocating the corresponding blocks on remote providers in a round-robin fashion.

We measure the write throughput for both HDFS and BSFS: the results can be seen on Figure 3(a). BSFS achieves a significantly higher throughput than HDFS, which is a result of the balanced, round-robin block distribution strategy used by BlobSeer. A high throughput is sustained by BSFS even when the file size increases (up to 16 GB). To evaluate of the load balancing in both HDFS and BSFS, we chose to compute the Manhattan distance to an ideally balanced system where all data providers/*datanodes* store the same number of blocks/chunks. To calculate this distance, we represent the data layout in each case by a vector whose size is equal to the number of data providers/*datanodes*; the elements of the vector represent the number of blocks/chunks stored by each provider/*datanode*. We compute 3 such vectors: one for HDFS, one for BSFS and one for a perfectly balanced system (where all elements have the same value: the total number of blocks/chunks divided by the total number of storage nodes. We then compute the distance between the "ideal" vector and the HDFS (respectively BSFS). As shown on Figure 3(b), as the file size (and thus, the number of blocks) increases, both BSFS and HDFS become unbalanced. However, BSFS remains much closer to a perfectly balanced system, and it manages to distribute the blocks almost evenly to the providers, even in the case of a large file. As far as we can tell, this can be explained by the fact that the block allocation policy in HDFS mainly takes into account data locality and does not aim at perfectly balancing the data distribution. A global load-balancing of the system is done for Map/Reduce applications when the tasks are assigned to nodes. During this experiment, we could notice that in HDFS there are *datanodes* that do not store any block, which explains the increasing curve

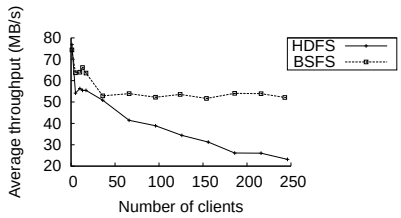


Figure 4: Performance of HDFS and BSFS when concurrent clients read from a single file

shown in figure 3(b). As we will see in the next experiments, a balanced data distribution has a significant impact on the overall data access performance.

4.3 Scenario 2: concurrent reads, shared file

In this scenario, for each given number N of clients varying from 1 to 250, we executed the experiment in two steps. First, we performed a boot-up phase, where a single client writes a file of $N \times 64$ MB, right after the deployment of HDFS/BSFS. Second, N clients read parts from the file concurrently; each client reads a different 64 MB chunk sequentially, using finer-grain blocks of 4 KB. This pattern where multiple readers request data in chunks of 4 KB is very common in the "map" phase of a Hadoop Map/Reduce application, where the mappers read the input file in order to parse the (key, value) pairs.

For this scenario, we ran two experiments in which we varied the data layout for HDFS. The first experiment corresponds to the case where the file read by all clients is entirely stored by a single *datanode* This corresponds to the case where the file has previously been entirely written by a client colocated with a *datanode* (as explained in the previous scenario). Thus, all clients subsequently read the data stored by one node, which will lead to a very poor performance of HDFS. We do not represent these results here. In order to achieve a more fair comparison where the file is distributed on multiple nodes both in HDFS and in BSFS, we chose to execute a second experiment. Here, the boot-up phase is performed on a dedicated node (no *datanode* is deployed on that node). By doing so, HDFS will spread the file in a more balanced way on multiple remote *datanodes* and the reads will be performed remotely for both BSFS and HDFS. This scenario also offers an accurate simulation of the first phase of a Map/Reduce application, when the mappers are assigned to nodes. The HDFS job scheduler tries to assign each map task to the node that stores the chunk the task will process; these tasks are called *local maps*. The scheduler also tries to achieve a global load-balancing of the system, therefore not all the assignments will be local. The tasks running on a different node than the one storing its input data, are called *remote maps*: they will read the data remotely.

The results obtained in the second experiment are presented on Figure 4. BSFS performs significantly better than HDFS, and moreover, it is able to deliver the same throughput even when the number of clients increases. This is a direct consequence of how balanced is the block distribution for that file.

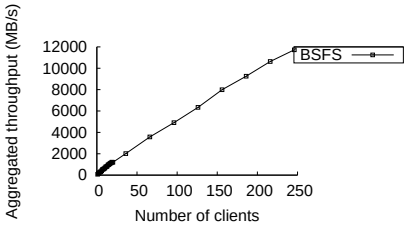


Figure 5: Performance of BSFS when concurrent clients append to the same file

The superior load balancing strategy used by BlobSeer when writing the file has a positive impact on the performance of concurrent reads, whereas the HDFS suffers from the poor distribution of the file chunks.

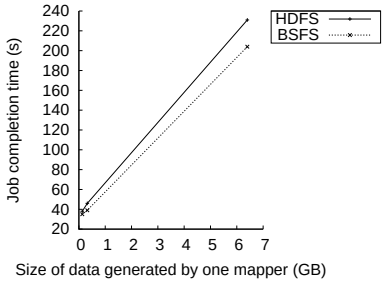
4.4 Scenario 3: Concurrent appends, shared file

We now focus on another scenario, where concurrent clients append data to the same file. This scenario is also useful in the context of Map/Reduce applications, as it is for a wide range of data-intensive applications in general. For instance, the possibility of running concurrent appends can improve the performance of a simple operation such as copying a large distributed file. This can be done in parallel by multiple clients which read different parts of the file, then concurrently append the data to the destination file. Moreover, if concurrent append operations are enabled, Map/Reduce workers can write the output of the reduce phase to the same file, instead of creating many output files, as it is currently done in Hadoop.

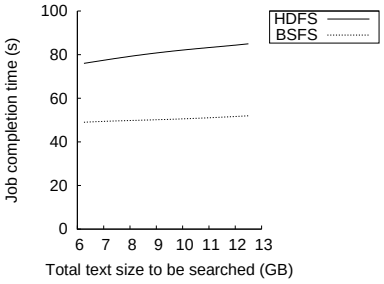
Despite its obvious usefulness, this feature is not available with Hadoop's file system: Hadoop has not been optimized for such a scenario. As BlobSeer provides support for efficient, concurrent appends by design, we have implemented the append operation in BSFS and evaluated the aggregated throughput as the number of clients varies from 1 to 250. We could not perform the same experiment for HDFS, since it does not implement the append operation.

Figure 5 illustrates the aggregated throughput obtained when multiple clients concurrently append data to the same BSFS file. These good results can be obtained thanks to BlobSeer, which is optimized for concurrent appends.

Note that these results also give an idea about the performance of concurrent writes to the same file. In BlobSeer, the append operation is implemented as a special case of the write operation where the write offset is implicitly equal to the current file size: the underlying algorithms are actually identical. The same experiment performed with writes instead of appends, leads to very similar results.



(a) RandomTextWrite: Job completion time for a total of 6.4 GB of output data when increasing the data size generated by each mapper



(b) Distributed grep: Job completion time when increasing the size of the input text to be searched

Figure 6: Benefits of using BSFS instead of HDFS as a storage layer in Hadoop: impact on the performance of Map/Reduce applications

4.5 Higher-level experiments with Map/Reduce applications

In order to evaluate how well BSFS and HDFS perform in the role of storage layers for real Map/Reduce applications, we selected two standard Map/Reduce applications that are part of Yahoo!'s Hadoop release.

The first application, *RandomTextWriter*, is representative of a distributed job consisting in a large number of tasks each of which needs to write a large amount of output data (with no interaction among the tasks). The application launches a fixed number of mappers, each of which generates a huge sequence of random sentences formed from a list of predefined words. The reduce phase is missing altogether: the output of each of the mappers is stored as a separate file in the file system. The access pattern generated by this application corresponds to concurrent, massively parallel writes, each of them writing to a different file.

To compare the performance of BSFS vs. HDFS in such a scenario, we co-deploy a Hadoop *tasktracker* with a *datanode* in the case of HDFS (with a data provider in the case of BSFS) on the same physical machine, for a total of 50 machines. The other entities for Hadoop, HDFS (namenode, *jobtracker*) and for BSFS (version manager, provider manager, namespace manager) are deployed on separate dedicated nodes. For BlobSeer, 10 metadata providers are deployed on dedicated machines as well.

We fix the total output size of the job to amount to 6.4 GB worth of generated text and vary the size generated by each mapper from 128 MB (corresponding to 50 parallel mappers) to 6.4 GB (corresponding to a single mapper), and measure the job completion time in each case.

Results obtained are displayed on Figure 6(a). Observe the relative gain of BSFS over HDFS ranges from 7 % for 50 parallel mappers to 11 % for a single mapper. The case of a single mapper clearly favours BSFS and is consistent with our findings for the synthetic benchmark in which we explained the respective behavior of BSFS and HDFS when a single process writes a huge file. The relative difference is smaller than in the case of the synthetic benchmark because here the total job execution time includes some computation time (generation

of random text). This computation time is the same for both HDFS and BSFS and takes a significant part of the total execution time.

The second application we consider is *distributed grep*. It is representative of a distributed job where huge input data needs to be processed in order to obtain some statistics. The application scans a huge text input file for occurrences of a particular expression and counts the number of lines where the expression occurs. Mappers simply output the value of these counters, then the reducers sum up the all the outputs of the mappers to obtain the final result. The access pattern generated by this application corresponds to concurrent reads from the same shared file.

In this scenario we co-deploy a *tasktracker* with a HDFS *datanode* (with a BlobSeer data provider, respectively), on a total of 150 nodes. We deploy all centralized entities (version manager, provider manager, namespace manager, *namenode*, etc) on dedicated nodes. Also, 20 Metadata providers are deployed on dedicated nodes for BlobSeer.

We first write a huge input file to HDFS and BSFS respectively. In the case of HDFS, the file is written from a node that is not colocated with a *datanode*, in order to avoid the scenario where HDFS writes all data blocks locally. This gives HDFS the chance to perform some load-balancing of data blocks. Then we run the distributed grep Map/Reduce application and measure the job completion time. We vary the size of the input file from 6.4 GB to 12.8 GB in increments of 1.6 GB. Since a Hadoop data block is 64 MB large and since usually Hadoop assigns a single mapper to process such a data block, this roughly corresponds to varying the number of concurrent mappers from 100 to 200.

Results obtained are represented in Figure 6(b). As can be observed BSFS outperforms HDFS by 35 % for 6.4 GB and the gap steadily increases to 38 % for 12.8 GB. This behavior is consistent with the results obtained for the synthetic benchmark where concurrent processes read from the same file. Again, the relative difference is smaller than in the synthetic benchmark because the job completion time accounts for both the computation time and the I/O transfer time. Note however the high impact of I/O in such applications that scan through the data for specific patterns: the benefits of supporting efficient concurrent reads from the same file at the level of the underlying distributed file system are definitely significant.

5 Conclusion

The efficiency of the Hadoop framework is a direct function of that of its data storage layer. This work demonstrates that it is possible to enhance it by replacing the default Hadoop Distributed File System (HDFS) layer by another layer, built along different design principles. We introduce our BlobSeer system, which is specifically optimized toward efficient, fine-grain access to massive, distributed data accessed under heavy concurrency. Thank to this new BlobSeer-based File System (BSFS) layer, the sustained throughput of Hadoop is significantly improved in scenarios that exhibit highly concurrent accesses to shared files. Moreover, BSFS supports additional features such as efficient concurrent appends, concurrent writes at random offsets and versioning. These features could be leveraged to extend or improve functionalities in future versions of Hadoop or other Map/Reduce frameworks. We list below several interesting perspectives.

Leveraging versioning. Although in most real Map/Reduce applications, data is mostly appended rather than overwritten, Hadoop's file system API does not implement append. Since BlobSeer supports arbitrarily concurrent writes as well as appends, this opens a high potential for very promising improvements of Map/Reduce framework implementations, including Hadoop. Versioning can be leveraged to optimize more complex Map/Reduce workflows, in which the output of one Map/Reduce is the input of another. In many such scenarios, datasets are only locally altered from one Map/Reduce pass to another: writing parts of the dataset while still being able to access the original dataset (thanks to versioning) could save a lot of temporary storage space.

Fault tolerance. An important aspect we did not discuss in this paper is fault tolerance. For this, we currently rely on classical mechanisms. At data level, we employ a simple replication mechanism that allows the user to specify a replication level for each BLOB. A write operation actually writes its respective blocks to a number of providers equal to that replication level. The metadata is stored in a DHT (formed by the metadata providers), which is resilient to faults by construction. The centralized managers represent single points of failure as is the case with the *namenode* of HDFS. Overall, fault-tolerance schemes currently used in BlobSeer are however rather minimal. We are currently exploring ways to replace them with distributed, fault-tolerant mechanisms, while still preserving a high-throughput for data access.

Security. We did not address security issues in this paper, as most of the time Hadoop deployments are exploited within private, trusted clusters owned by big companies, such as Google and Yahoo!: for now, we place ourselves in the same context, therefore the security assumptions are basically the same as for Hadoop's built-in file system. In the case where Hadoop would run as a Map/Reduce cloud service, possibly relying on externalized, virtualized resources from other cloud computing service providers (such as Amazon), the security constraints would be different. It then becomes crucial to guarantee data privacy and data access control for multiple users, according to a contract. We plan to explore these issues in the near future.

References

[1] Amazon Elastic Compute Cloud (EC2). http://aws.amazon.com/ec2/.

[2] Amazon Elastic Map Reduce. http://aws.amazon.com/elasticmapreduce/.

[3] Amazon Simple Storage Service (S3). http://aws.amazon.com/s3/.

[4] Kosmix CloudStore Distributed File System. http://kosmosfs.sourceforge.net/index.html.

[5] Jeffrey Dean and Sanjay Ghemawat. MapReduce: simplified data processing on large clusters. *Communications of the ACM*, 51(1):107–113, 2008.

[6] Sanjay Ghemawat, Howard Gobioff, and Shun-Tak Leung. The google file system. *SIGOPS - Operating Systems Review*, 37(5):29–43, 2003.

[7] The Apache Hadoop Project. http://www.hadoop.org.

[8] HDFS. The Hadoop Distributed File System. http://hadoop.apache.org/common/docs/r0.20.1/hdfs_design.html.

[9] 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.

[10] Bogdan Nicolae, Gabriel Antoniu, and Luc Bougé. BlobSeer: How to enable efficient versioning for large object storage under heavy access concurrency. In *Proc. 2nd Workshop on Data Management in Peer-to-Peer Systems (DAMAP'2009)*, Saint Petersburg, Russia, March 2009. Held in conjunction with EDBT'2009.

[11] Bogdan Nicolae, Gabriel Antoniu, and Luc Bougé. Enabling high data throughput in desktop grids through decentralized data and metadata management: The blobseer approach. In *Proc. 15th International Euro-Par Conference on Parallel Processing (Euro-Par '09)*, volume 5704 of *Lect. Notes in Comp. Science*, pages 404–416, Delft, The Netherlands, 2009. Springer-Verlag.

[12] PVFS. Parallel virtual file system, version 2. http://pvfs2.org/.

[13] Frank B. Schmuck and Roger L. Haskin. GPFS: A shared-disk file system for large computing clusters. In *FAST '02: Proceedings of the Conference on File and Storage Technologies*, pages 231–244. USENIX Association, 2002.

[14] Garth Gibson Wittawat Tantisiriroj, Swapnil Patil. Data-intensive file systems for internet services: A rose by any other name... Technical Report UCB/EECS-2008-99, Parallel Data Laboratory, October 2008.

[15] C. Zheng, G. Shen, S. Li, and S. Shenker. Distributed Segment Tree: Support range query and cover query over DHT. In *Proceedings of the Fifth International Workshop on Peer-to-Peer Systems (IPTPS)*, Santa Barbara, California, 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