Towards Scalable Array-Oriented Active Storage: the Pyramid Approach

Viet-Trung Tran, Bogdan Nicolae, Gabriel Antoniu

Towards Scalable Array-Oriented Active Storage: the Pyramid Approach

Viet-Trung Tran
ENS Cachan - Brittany
Rennes, France
Viet-trung.Tran@irisa.fr

Bogdan Nicolae
INRIA
Saclay - Île-de-France, France
Bogdan.Nicolae@inria.fr

Gabriel Antoniu
INRIA
Rennes - Bretagne Atlantique, France
Gabriel.Antoniu@inria.fr

ABSTRACT

The recent explosion in data sizes manipulated by distributed scientific applications has prompted the need to develop specialized storage systems capable to deal with specific access patterns in a scalable fashion. In this context, a large class of applications focuses on parallel array processing: small parts of huge multi-dimensional arrays are concurrently accessed by a large number of clients, both for reading and writing. A specialized storage system that deals with such an access pattern faces several challenges at the level of data/metadata management. We introduce Pyramid, an active array-oriented storage system that addresses these challenges. Experimental evaluation demonstrates substantial scalability improvements brought by Pyramid with respect to state-of-art approaches both in weak and strong scaling scenarios, with gains of 100% to 150%.

1. INTRODUCTION

We are witnessing a rapidly increasing number of application areas generating and processing very large volumes of data on a regular basis. Such applications, called *data-intensive*, are now present in a variety of disciplines including astronomy, biology, physics, oceanography, atmospheric sciences, where the amounts of data to be managed have reached the exa-scale. In this context, the scalability of data management becomes a critical issue, as it affects the performance of the whole application.

Many established storage solutions such as parallel file systems and database management systems strive to achieve high-performance at large-scale, however one major difficulty is to achieve performance scalability of data accesses under concurrency. One limitation comes from the fact that most existing solutions expose data access models (e.g. file systems, structured databases) that are too general and do not exactly match the natural requirements of the application. This forces the application developer to either adapt to the exposed data access model or to use an intermediate layer that performs a translation. In either case, this mismatch leads to suboptimal data management: as noted in [16], the one-storage-solution-fits-all-needs has reached its limits.

The situation described above highlights an increasing need to specialize the I/O stack to match the requirements of the applications. In scientific computing, of particular interest is a large class of applications that represent and manipulate data as huge multi-dimensional arrays [13]. Such applications typically consist of a large number of distributed workers that concurrently process subdomains of those arrays. This context highlights two important requirements. The first one is the ability of the I/O system to sustain a high throughput for parallel subdomain processing. The second important requirement in this context regards versioning: according to [15], scientific applications often need to access past snapshots of the data. This can be useful in several scenarios that could be identified in data-intensive applications. One possible goal is to provide an efficient means for visualizing a phenomenon as it progresses in time. Another example is to enable efficient data post-processing, for instance to compute statistics on the evolution of that phenomenon.

In this paper we address both requirements mentioned above. We propose *Pyramid*, a specialized array-oriented storage manager optimized for parallel array processing.

Relationship to authors' previous work. Pyramid is inspired by BlobSeer [11, 17], a versioning-oriented storage system specifically optimized to sustain a high throughput under concurrency. BlobSeer focuses on unstructured Binary Large OBjects (BLOBs) that represent linear addressing spaces, whereas Pyramid builds on similar design principles to specifically address the case multi-dimensional data. Pyramid also introduces several new features. A preliminary description and evaluation of Pyramid was first introduced in [18]. This paper extends [18] with more detailed descriptions and more thorough experiments.

Contributions. Our contributions can be summarized as follows: (1) we introduce a dedicated array-oriented data access model that offers support for active storage and versioning; (2) we enrich striping techniques specifically optimized for multi-dimensional arrays [12] with a distributed metadata management scheme that avoids potential I/O bottlenecks observed with centralized approaches; (3) we evaluate Pyramid for highly concurrent data access patterns, comple-

menting weak scalability results reported in [18] with new experiments that evaluate strong scalability.

2. RELATED WORK

SciDB [3, 15] is an effort to design an open-source database system for scientific data analytics. SciDB departs from the relational database model, offering an array-oriented database model that specifically targets multi-dimensional data. Notably, SciDB proposes two features that are crucial in scientific data analytics: multi-dimensional aware data striping and data versioning. However, a concrete solution to implement these features is still on-going work. Furthermore, with the upcoming exa-scale age, scalable metadata management becomes a critical issue that is insufficiently addressed by SciDB. Our approach aims to address these limitations.

Emad et al. introduced ArrayStore [14], a storage manager for complex, parallel array processing. Similarly to SciDB, ArrayStore partitions large arrays into chunks and stores them in a distributed fashion. ArrayStore organizes metadata as R-trees, which are maintained in a centralized fashion. At large scale, this inevitably leads a bottleneck with respect to the scalability of metadata access, because it places a limit on the number of metadata queries that can be served concurrently. Furthermore, ArrayStore is only designed as a read-only storage system. The authors acknowledge that ArrayStore was not optimized to handle updates to multi-dimensional data: this scenario can cause significant performance degradation due to design focus on read performance. Unlike ArrayStore, our Pyramid approach is designed to efficiently support workloads that consist of any mix of concurrent reads and writes.

3. REQUIREMENTS

In this section we discuss the main challenges and requirements that array-oriented storage systems need to address in our view.

Specialized array-oriented storage model. Traditionally, the I/O stack used on high performance computing infrastructure consists of three layers. The lowest layer is the parallel file system. It is responsible to aggregate the storage space of dedicated storage resources in a single common pool, which is exposed to higher layers using a file access model, typically POSIX [2]. This file access model is leveraged by the I/O middleware (such as MPI-I/O [6]), the layer directly on top of the parallel file system. It is specifically designed to to coordinate and optimize the parallel access patterns of HPC applications, acting as a bridge between these recurring patterns and the more generic file-oriented I/O access model. At this level, data is still in a raw form and has no structure attached to it. Since most HPC-oriented applications do not directly work with raw data but tend to associate a multi-dimensional structure to it, a third layer in form of an I/O library (such as [9, 1, 10]) provides the necessary mechanisms to attach structure to the data an to perform multi-dimensional queries and updates to it.

A major problem with this traditional three-layered I/O stack is the existence of a mismatch between the access models expected at the lower layers. The I/O of the application is funneled from the highest layer down to the parallel file system, at each step going through a translation process that is necessary in order to adapt to the expected access model. These access models (i.e. POSIX and MPI-I/O) and designed to handle the worst-case scenarios for conflicts, synchronization, and coherence of data that is represented in a linear fashion, ignoring the original structure and purpose of the I/O. Thus, the translation process between the layers becomes very costly and incurs a major performance overhead. Although a stack-based design has its advantages (i.e. it is easier to design and to develop independently), in [8] it has been pointed out that it has limited potential for scalability at exa-scale.

To alleviate this problem, we argue in favor of a specialized storage system that is designed from scratch to directly match the I/O needs of the application. Such a design effectively "shortcuts" through the I/O stack and gains direct control over the storage resources, setting it free from any unnecessary constraints and enabling it to take more informed decisions, which in turn provides better optimization opportunities.

Versioning. Data versioning is a feature that is increasingly needed by users of scientific applications, because of several reasons.

First, it provides a convenient tool to explore the history of changes to a dataset and at the same time avoids unnecessary data duplication by saving incremental differences only. This is a scenario frequently encountered during the simulation of scientific phenomena, where the result of each iteration usually corresponds to the changes in time observed during the simulation. In such scenarios, changes are often localized and affect only small parts of the data, making a full output of the state at each iteration both inefficient and difficult to track.

Second, versioning facilitates sharing of data sets among multiple users. For example, consider a globally shared data set that is used as input for a series of experiments, each of which is performed by a different scientist that is interested to manipulate the data in a different fashion. In such a scenario, users want their own view of the data set. A simple solution to this problem is to create a full copy of the data set for each user. However, such an approach is expensive both in terms of performance and storage utilization. Using versioning, the illusion of a dedicated copy can be easily created for each user, while internally optimizing performance and resource utilization.

Finally, versioning is a key tool that enables keeping track of data provenance [5]. This is becoming a critical issue in scientific communities, as external data sources are increasingly relied upon as an input to scientific applications. In this context, it is important to be able to track the whole I/O workflow that produced a data set through all transformations, analyzes, and interpretations, which enables better management, sharing and reuse of data in a reliable fashion.

High performance under concurrent access. To process a huge amount of data in a timely fashion, scientific applications are usually distributed at large scale. With increasing scale, concurrent I/O operations on shared data sets suffer performance degradation. This effect is particularly noticeable when data needs not only be read, but also written. Such workloads are increasingly noticeable in scientific applications, as they gain in complexity and funnel the data thorough more elaborate workflows.

Thus, it is important to design a storage system that is able to sustain a high data access throughputs under concurrency, both for reading and writing.

4. GENERAL DESIGN

Our approach relies on a series of key design principles:

Array versioning. At the core of our approach is the idea of representing data updates using immutable data and metadata. Whenever a multi-dimensional array needs to be updated, the affected cells are never overwritten, but rather a new snapshot of the whole array is created, into which the update is applied. In order to offer the illusion of fully-independent arrays with minimal overhead in both storage space utilization and performance, we rely on differential updates: only the modified cells are stored for each new snapshot. Any unmodified data or metadata is shared between the new snapshot and the previous snapshots. The metadata of the new snapshot is generated in such way that it seamlessly interleaves with the metadata of the previous snapshots to create incremental snapshots that look and act at application level as independent arrays.

Active storage support. Many datacenters nowadays are made of machines equipped with commodity hardware that often act as both storage elements and compute elements. In this context, it is highly desirable to be able to move the computation to the data rather than the other way around, for two reasons: (1) it conserves bandwidth, which is especially important when data transfers are expensive (e.g. because of cost concerns or because of high latency / low bandwidth); (2) it enables better workload parallelization, as part of the work can be delegated to the storage elements (e.g. post-processing filters, compression, etc.).

Versioning array-oriented access interface. We propose an interface to multi-dimensional data that is specifically designed to enable fine-grained versioning access to subdomains while offering the features mentioned above.

id = CREATE(n, sizes[], defval)

creates a n-dimensional array identified by id and spanning sizes[i] cells in each dimension $0 \leq i < n$. By convention, the initial snapshot associated to the array has version number 0 and all cells are filled with the default initial value defval. This is a lazy initialization: no data and metadata is added until some cells of the array are actually updated.

READ(id, v, offsets[], sizes[], buffer)

reads a subdomain from the snapshot v of the array id. The subdomain is delimited by offsets[i] and spans sizes[i] cells in each dimension $0 \leq i < n$. The contents of the subdomain is stored in the local memory region buffer.

w = WRITE(id, offsets[], sizes[], buffer)

writes the content of the local memory region buffer to the cells of the subdomain delimited by offsets[i] and sizes[i] in each dimension $0 \leq i < n$ of the array id. The result is a new snapshot whose version number is w.

w = SEND_COMPUTATION(id, v, offsets[], sizes[], f)

applies function f to all cells of the subdomain delimited by offsets[i] and sizes[i] in each dimension $0 \leq i < n$. The result is a new snapshot whose version number is w. The computation is performed remotely on the storage elements and involves no additional data transfers.

Multi-dimensional aware data chunking. Chunking is a standard approach to reduce contention for parallel accesses to multi-dimensional data [12]. The core idea is very simple: split the array into chunks and distribute them among the storage elements, which results in a distribution of the I/O workload.

In this context, the partitioning scheme plays a crucial role: under unfavorable conditions, read and write queries may generate "strided" access patterns (i.e. access small parts from a large number of chunks), which has a negative impact on performance. To reduce this effect, we propose to split the array into subdomains that are equally sized in each dimension. Using this approach, the neighbors of cells have a higher chance of residing in the same chunk irrespective of the query type, which greatly limits the number of chunks that need to be accessed.

Furthermore, this scheme brings an important advantage for active storage: since data is distributed among multiple storage elements, so does any computation that is sent to the data, leading to an efficient implicit parallelization.

Lock-free, distributed chunk indexing. Data is striped and stored in a distributed fashion, which implies that additional metadata is necessary to describe the composition of arrays in terms of chunks and where these chunks can be found.

The problem of building spatial indexes has been studied extensively, with several specialized data structures proposed: R-trees, xd-trees, quad-trees, etc. Most of these structures were originally designed and later optimized for centralized management.

However, a centralized metadata management scheme limits scalability as in distributed file systems. Even in the situation when enough storage is available for storage of metadata, the metadata server can quickly become a bottleneck when attempting to serve a large number of clients simultaneously.

Thus, it is important to implement a distributed metadata management scheme. To this end, we propose a distributed quad-tree like structure that is used to index the chunk lay-

out and is specifically optimized for concurrent updates. Our scheme takes advantage of the fact that data and metadata remains immutable in order to efficiently handle concurrent metadata updates without locking.

5. PYRAMID

This section introduces *Pyramid*, a complete storage solution for multi-dimensional data that is based on the design principles presented in Section 4.

5.1 Architecture

Pyramid is a distributed system consisting of the following components:

Version managers are the core of Pyramid. They coordinate the process of assigning new snapshot versions for concurrent writes such that total ordering is guaranteed. At the same time, they wait for the moment when snapshots are consistent and expose them to the readers in an atomic fashion. Pyramid can be configured to use multiple version managers that collaborate to achieve the aforementioned objectives in a distributed fashion. This design favors scalability and fault tolerance over centralized approaches.

Metadata managers implement the distributed quad-trees introduced in the previous section. They are responsible for instructing the clients what chunks to fetch from what location for any given subdomain.

Active storage servers physically store chunks generated by creating new arrays or updating existing arrays. An active storage server can also execute handler functions assigned to each object.

A storage manager is in charge of monitoring all available storage servers and schedule the placement of newly created chunks based on the monitoring information.

5.2 Zoom on chunk indexing

In Section 4 we argued in favor of a distributed chunk indexing scheme that leverages versioning to avoid potentially expensive locking under concurrency. In this section we briefly describe how to achieve this objective by introducing a distributed tree structure that is specifically designed to take advantage of the fact that data and metadata remains immutable.

Our solution generalizes the metadata management proposed in [11], which relies on the same principle to achieve high metadata scalability under concurrency. For simplicity, we illustrate our approach for a two-dimensional array in the rest of this section, a case that corresponds to a quad-tree (i.e. each inner node has four children). The same approach can be easily generalized for an arbitrary number of dimensions.

Structure of the distributed quad-tree. We make use of a partitioning scheme that recursively splits the initial two-dimensional array into four subdomains, each corresponding to one of the four quadrants: Upper-Left (UL), Upper-Right (UR), Bottom-Left (BL), Bottom-Right (BR). This process continues until a subdomain size is reached that is small enough to justify storing the entire subdomain as a single chunk. To each subdomain obtained in this fashion, we associate a tree node (said to "cover" the subdomain) as follows: the leaves cover single chunks (i.e. they hold information about what storage servers store the chunks), the inner nodes have four children and cover the subdomain formed by the quadrants (i.e. UL, UR, BL, BR), while the root covers the whole array.

All tree nodes are labeled with a version number (initially 0) that corresponds to the snapshot to which they belong. Updates to the array generate new snapshots with increasing version numbers. Inner nodes may have as children nodes that are labeled with a smaller version number, which effectively enables sharing of unmodified data and their whole corresponding sub-trees between snapshots. Figure 1 illustrates this for an initial version of the array to which an update is applied.

Since tree nodes are immutable, they are uniquely identified by the version number and the subdomain they cover. Based on this fact, we store the resulting tree nodes persistently in a distributed fashion, using a Distributed Hash Table (DHT) maintained by the metadata managers: for each tree node a corresponding key-value pair is generated and added. Thanks to the DHT, accesses to the quad-tree are distributed under concurrency, which eliminates metadata bottlenecks present in centralized approaches.

Read metadata. A read query descends into the quad tree in a top-down fashion: starting from the root, it recursively visits all quadrants that cover the requested subdomain of the read query until all involved chunks are discovered. Once this step has completed, the chunks are fetched from the corresponding storage servers and brought locally. Note that the tree nodes remain immutable, which enables reads to proceed in parallel with writes, without the need to synchronize quad tree accesses.

Write queries. A write query first sends the chunks to the corresponding storage servers and then builds the quad-tree associated to the new snapshot of the array. This is a bottom-up process: first the new leaves are added in the DHT, followed by the inner nodes up to the root. For each inner node, the four children are established: some may belong to earlier snapshots. Under a concurrent write access pattern, this scheme apparently implies a synchronization of the quad-tree generation, because of inter-dependencies between tree nodes. However, we avoid such a synchronization by feeding additional information about the other concurrent writers during the quad-tree generation. This enables each writer to "predict" what tree nodes will be generated by the other writers and use those tree nodes as children if necessary, under the assumption that the missing children will be eventually added to the DHT by the other writers.

Consistency semantics. Since readers always access snapshots explicitly specified by the snapshot version, they are completely separated from the writers. Writers do not access any explicitly specified snapshot but can be thought of accessing an implicit virtual snapshot, which intuitively represents the most recent view of the multi-dimensional domain. Concurrent writes are guaranteed to be atomic and

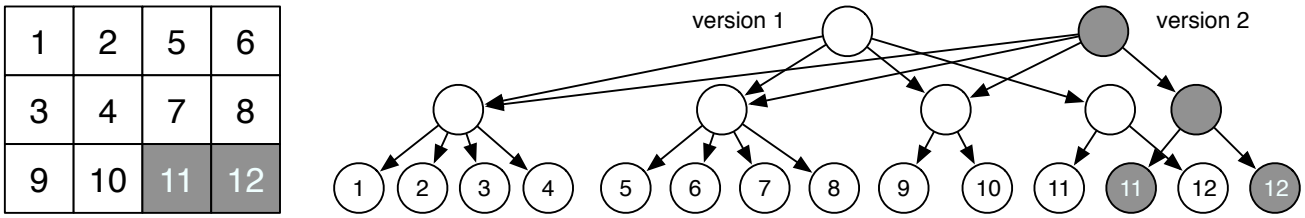

Figure 1: Metadata quad-trees by example: two chunks (dark color) of an initial array (partitioned according to the left figure) are updated, leading to the additional tree nodes and their links represented in the right figure.

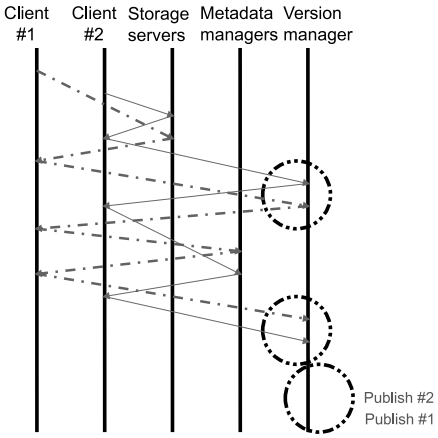

Figure 2: Total ordering of two concurrent updates: snapshots are published in the order in which the data was written in order to guarantee metadata consistency

totally ordered from the user point of view. This guarantee is enforced by the version managers, responsible to delay the publication of the new snapshots until the moment when all metadata is consistent readers can safely access the new snapshots.

An example of how this works is depicted in Figure 2. Client 2 finishes writing data faster than client 1 and thus generates a snapshot that precedes the snapshot of client 1. However, client 2 is slower at writing the metadata. Thus, the metadata of client 1 has dependencies on client 2 and neither of them can be safely accessed before client 2 finishes writing the metadata. When this happens, the version managers publish both the snapshot of client 1 and the snapshot of client 2 in an atomic fashion, effectively enabling the readers to access both snapshots.

6. EVALUATION

We evaluated Pyramid through a set of experiments on the Grid'5000 [7] testbed that aims to evaluate both the performance and the scalability of our approach under concurrent accesses. To this end, we simulated a common access pattern exhibited by scientific applications: 2D array dicing. This access pattern involves a large number of processes that read and write distinct parts of the same large array in parallel.

We focus on two settings: a weak scalability setting and a strong scalability setting. In the weak scalability setting, we keep the size of the subdomain that each client process accesses constant, while increasing the number of concurrent clients. In the strong scalability setting, we keep the total size of array constant while increasing the number of concurrent processes that access increasingly smaller parts of the array.

Each setting uses at most 140 nodes of the Graphene cluster. We dedicated 64 nodes to deploy our client processes while the rest of the nodes are used to deploy Pyramid in the following configuration: 1 version manager, 1 storage manager. We co-deployed on the 74 remaining nodes a metadata manager together with and active storage provider. We then compare Pyramid to the case when a standard parallel file system is used to store the whole array as a single sequence of bytes in a file. To this end, we deployed an instance of PVFS2 [4] on the same 76 nodes used to evaluate Pyramid.

Weak Scalability. In this setting, each process reads and writes a 1 GB large subdomain that consists of 32x32 chunks (i.e. 1024x1024 bytes for each chunk). We start with an array that holds a single such subdomain (i.e. it corresponds to a single process) and gradually increase its size to 2x2, 3x3, ... 7x7 subdomains (which corresponds to 4, 9, ... 49 parallel processes).

Results are shown in Figure 3. As can be observed, the throughput reaches 80 MB/s for one single client, demonstrating high performance even for fine granularity decompositions. Furthermore, with increasing number of concurrent clients, the aggregated throughput grows steadily up to 2.1 GB/s, which represents an increase of about 100% over PVFS. This demonstrates a much better scalability of our approach, which is a consequence of both the multi-dimensional aware data striping and the distributed metadata management. On the other hand, the scalability of PVFS2 suffers because the array is represented as a single sequence of bytes, which leads to fine-grain, strided access patterns.

Strong scalability. In this setting, the domain size is fixed at 64 GB (1024x1024 chunks). We start with one single process and gradually increase the number of processes up to 64 concurrent processes in a layout of 2x2, 2x4, 4x4, 4x8,

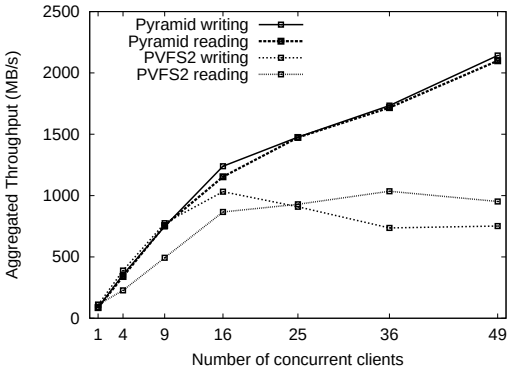

Figure 3: Weak scalability: fixed subdomain size, increasing number of client processes

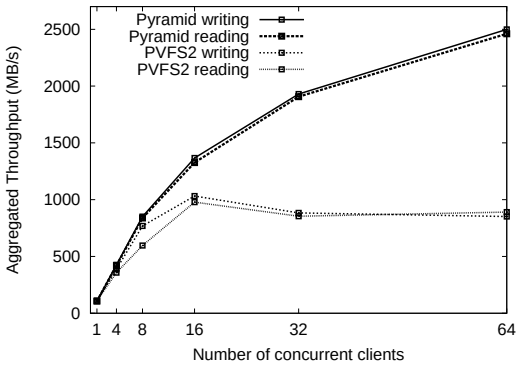

Figure 4: Strong scalability: fixed total domain size, increasing number of client processes

8x8. In this case, the size of the subdomain for each client depends on the number of concurrent processes.

As can be observed in Figure 4, our approach demonstrates much higher strong scalability than PVFS for both read and write workloads. More specifically, both Pyramid and PVFS sustain a similar throughput for 1 process, both for read and write workloads. However, when increasing the number of concurrent processes, unlike for our approach, the curves corresponding to PVFS rapidly flatten. In the extreme case of 64 concurrent processes, Pyramid is able to sustain a total aggregated throughput of more than 2.5 GB/s, which represents an increase of over 150% over PVFS.

7. CONCLUSIONS

We introduced Pyramid, an array-oriented storage system that offers support for *active storage* and *versioning*. Through striping techniques specifically optimized for multi-dimensional arrays combined with a distributed metadata management scheme, our approach addresses the I/O requirements of scalable I/O parallel processing and avoids the I/O bottlenecks observed with centralized approaches.

Preliminary evaluation shows promising results: our prototype demonstrates good performance and scalability under concurrency, both for read and write workloads. In terms of weak scalability of aggregated throughput, Pyramid outperforms PVFS by 100%. As regards strong scalability, the gain over PVFS reaches 150% for 64 concurrent processes, for a total aggregated throughput of more than 2.5 GB/s.

In future work, we plan to explore the possibility of using Pyramid as a storage backend for SciDB [3] and HDF5 [1]. This direction has a high potential to improve I/O throughput while keeping compatibility with standardized data access interfaces. Another promising direction for our approach are scientific applications that process arrays at different resolutions: many times whole subdomains (e.g. zero-filled regions) can be characterized by simple summary information. In this context, our distributed metadata scheme can be enriched to hold such summary information about the subdomains in the tree nodes, which can be relied upon to avoid deeper fine-grain accesses when possible.

Acknowledgments

This work was supported in part by the Agence Nationale de la Recherche (ANR) under Contract ANR-10-SEGI-01 (MapReduce Project). The experiments presented in this paper were carried out using the Grid'5000/ALADDIN-G5K experimental testbed, an initiative of the French Ministry of Research through the ACI GRID incentive action, INRIA, CNRS and RENATER and other contributing partners (see http://www.grid5000.fr/).

8. REFERENCES

[1] Hdf5. http://www.hdfgroup.org/about/hdf_technologies.html.

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

[3] P. Brown. Overview of SciDB: large scale array storage, processing and analysis. In *SIGMOD '10: Proceedings of the 2010 International conference on Management of data*, pages 963–968, Indiana, USA, 2010. ACM.

[4] P. H. Carns, W. B. Ligon, R. B. Ross, and R. Thakur. PVFS: A parallel file system for Linux clusters. In *Proceedings of the 4th Annual Linux Showcase and Conference*, pages 317–327, Atlanta, GA, 2000. USENIX Association.

[5] J. Cheney, L. Chiticariu, and W.-C. Tan. Provenance in databases: Why, how, and where. *Foundations and trends in databases*, 1:379–474, April 2009.

[6] R. L. Graham. The MPI 2.2 Standard and the Emerging MPI 3 Standard. In *EuroMPI '09: Proceedings of the 16th European PVM/MPI Users' Group Meeting on Recent Advances in Parallel Virtual Machine and Message Passing Interface*, pages 2–2, Espoo, Finland, 2009.

[7] 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*, 20(4):481–494, November 2006.

[8] S. Lang, P. Carns, R. Latham, R. Ross, K. Harms, and W. Allcock. I/o performance challenges at leadership

scale. In *SC '09: Proceedings of the Conference on High Performance Computing Networking, Storage and Analysis*, pages 40:1–40:12, Portland, USA, 2009.

[9] J. Li, W.-k. Liao, A. Choudhary, R. Ross, R. Thakur, W. Gropp, R. Latham, A. Siegel, B. Gallagher, and M. Zingale. Parallel netcdf: A high-performance scientific i/o interface. In *SC '03: Proceedings of the 2003 ACM/IEEE conference on Supercomputing*, pages 39–47, Phoneix, USA, 2003.

[10] J. F. Lofstead, S. Klasky, K. Schwan, N. Podhorszki, and C. Jin. Flexible io and integration for scientific codes through the adaptable io system (adios). In *CLADE '08: Proceedings of the 6th international workshop on Challenges of large applications in distributed environments*, pages 15–24, Boston, USA, 2008.

[11] 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*, 71:169–184, February 2011.

[12] S. Sarawagi and M. Stonebraker. Efficient organization of large multidimensional arrays. In *ICDE '94: Proceedings of the 10th International Conference on Data Engineering*, pages 328–336, Houston, USA, 1994.

[13] E. Smirni, R. Aydt, A. Chien, and D. Reed. I/O requirements of scientific applications: An evolutionary view. In *HPDC '02: Proceedings of 11th IEEE International Symposium on High Performance Distributed Computing*, pages 49–59, Edinburgh, UK, 2002. IEEE.

[14] E. Soroush, M. Balazinska, and D. Wang. Arraystore: a storage manager for complex parallel array processing. In *SIGMOD '11: Proceedings of the 2011 International conference on management of data*, pages 253–264, Athens, Greece, 2011. ACM.

[15] M. Stonebraker, J. Becla, D. Dewitt, K.-T. Lim, D. Maier, O. Ratzesberger, and S. Zdonik. Requirements for Science Data Bases and SciDB. In *CIDR '09: Proceedings of the 4th Conference on Innovative Data Systems Research*, 2009.

[16] M. Stonebraker and U. Cetintemel. One size fits all: An idea whose time has come and gone. In *ICDE '05: Proceedings of the 21st International Conference on Data Engineering*, pages 2–11, Tokyo, Japan, 2005.

[17] V.-T. Tran, B. Nicolae, G. Antoniu, and L. Bougé. Efficient support for MPI-I/O atomicity based on versioning. In *CCGrid 2011: Proceedings of the 11th IEEE/ACM International Symposium on Cluster, Cloud, and Grid Computing*, pages 514–523, Newport Beach, USA, 2011.

[18] V.-T. Tran, B. Nicolae, G. Antoniu, and L. Bougé. Pyramid: A large-scale array-oriented active storage system. In *LADIS '11: Proceedings of the 5th Workshop on Large-Scale Distributed Systems and Middleware*, Seattle, USA, 2011.