Mission Possible: Unify HPC and Big Data Stacks Towards Application-Defined Blobs at the Storage Layer

Pierre Matri, Yevhen Alforov, Alvaro Brandon, María Pérez, Alexandru Costan, Gabriel Antoniu, Michael Kuhn, Philip Carns, Thomas Ludwig

Mission Possible: Unify HPC and Big Data Stacks Towards Application-Defined Blobs at the Storage Layer

Pierre Matri[a], Yevhen Alforov[b], Álvaro Brandon[a], María S. Pérez[a], Alexandru Costan[c], Gabriel Antoniu[c], Michael Kuhn[d], Philip Carns[e], Thomas Ludwig[b]

[a] *Universidad Politécnica de Madrid, Madrid, Spain*
[b] *Deutsches Klimarechenzentrum, Hamburg, Germany*
[c] *Inria Rennes, France*
[d] *Universität Hamburg, Hamburg, Germany*
[e] *Argonne National Laboratory, Lemont, IL, USA*

Abstract

HPC and Big Data stacks are completely separated today. The storage layer offers opportunities for convergence, as the challenges associated with HPC and Big Data storage are similar: trading versatility for performance. This motivates a global move towards dropping file-based, POSIX-IO compliance systems. However, on HPC platforms this is made difficult by the centralized storage architecture using file-based storage. In this paper we advocate that the growing trend of equipping HPC compute nodes with local storage redistributes the cards by enabling object storage to be deployed alongside the application on the compute nodes. Such integration of application and storage not only allows fine-grained configuration of the storage system, but also improves application portability across platforms. In addition, the single-user nature of such application-specific storage obviates the need for resource-consuming storage features like permissions or file hierarchies offered by traditional file systems. In this article we propose and evaluate Blobs (Binary Large Objects) as an alternative to distributed file systems. We factually demonstrate that it offers drop-in compatibility with a variety of existing applications while improving storage throughput by up to 28%.

1. Introduction

HPC and Big Data platforms are carving new data storage models. This is made necessary by the ever-increasing scale of the computation and of the datasets ingested and produced by large-scale applications. The success of key-value stores [1, 2] or block storage systems [3, 4] on Clouds, and the advent of burst buffers [5, 6] or advanced I/O libraries [7, 8] for HPC clearly highlight this need.

At the heart of these different methods is the move from legacy POSIX-compliant storage systems towards simple storage paradigms designed especially for one purpose, trading versatility for performance. Indeed, POSIX-IO imposes functionality such as hierarchical namespaces or file permissions. While these features are often provided for convenience, they are in practice rarely needed by modern applications and can significantly hinder the storage performance. Indeed, the libraries and frameworks commonly used to access the storage on HPC [9] and Big Data platforms [10, 11] provide relaxed semantics (*i.e.*, the set of rules and guarantees provided by the system regarding the behavior of its storage operations) compared to those of the underlying file system.

Yet, deploying new storage models on HPC platforms used to be hard or simply impossible. Indeed, parallel file systems such as Lustre or GPFS on HPC have been the cornerstone of HPC storage for decades and are likely to remain so in the next few years. This is largely explained by the high level of versatility and support for legacy applications, which is without comparison with that of purpose-built storage systems. In contrast, this is easy on cloud computing platforms such as [12, 13], which enable users to deploy and configure exactly the storage system they need on compute nodes.

Email addresses: `pmatri@fi.upm.es` (Pierre Matri), `alforov@dkrz.de` (Yevhen Alforov), `abrandon@fi.upm.es` (Álvaro Brandon), `mperez@fi.upm.es` (María S. Pérez), `alexandru.costan@irisa.fr` (Alexandru Costan), `gabriel.antoniu@inria.fr` (Gabriel Antoniu), `michael.kuhn@informatik.uni-hamburg.de` (Michael Kuhn), `carns@mcs.anl.gov` (Philip Carns), `ludwig@dkrz.de` (Thomas Ludwig)

Application-defined storage on HPC comes as a solution. It leverages local storage on compute nodes available in a growing number of leadership-class supercomputers [14, 15] to allow scientists to deploy transient data services alongside the application. Such services offer the application exactly the semantics and fine-grained tuning it needs. Multiple examples of such services exist in the literature [16, 17]. This integration of the application with the storage it needs greatly eases its containerization and hence application portability across platforms.

We argue that deploying storage alongside the application additionally obviates the need for the aforementioned features of distributed file systems by removing the multi-user constraint. Blob-, or Object-based storage [18, 19, 20] has been demonstrated to provide an alternative to file-based storage on HPC and Big Data platforms. The reason is twofold. First, the flat namespace and simple semantics they provide enables performance improvements that are simply inaccessible to distributed file systems. Second, the data model they provide is close enough to that of file systems so most applications could use it with little to no modification.

In this article we leverage application-defined storage to assess the applicability and benefits of object-based storage for both HPC and Big Data platforms. Our contributions can be summarized as follows:

- After briefly describing our goals and reviewing related work (Section 2), **we propose blobs as candidates for addressing the storage needs of HPC and Big Data** (Section 3).
- We leverage a representative set of HPC and Big Data applications to prove that the vast majority of the **I/O calls performed can be covered by state-of-the-art blob storage systems** (Section 6).
- We **describe the modifications necessary in the storage stack** for HPC and Big Data applications to run atop blob storage (Section 7).
- We use an experimental testbed and a leadership-class supercomputer (Section 5) to **evaluate the performance benefits and trade-offs running these applications atop the same blob storage systems** rather than traditional file-based storage (Section 8). We highlight a completion time improvement of up to 25% with blobs.

We finally conclude with future work that enhances our proposal (Section 9).

2. Related work and motivation

In this section we start by reviewing the state of the art regarding relaxing POSIX semantics on both HPC and Big Data applications, convergence between both these worlds and application-defined storage for HPC.

2.1. HPC: Relaxing POSIX-IO API and semantics

Increasingly large amounts of data are generated by HPC applications as the result of simulations and large-scale experiments. Thus, storage systems need to provide concurrent access to the data for large numbers of tasks and processes. Such parallel storage operations rely on the usage of a parallel file system (PFS) implementing the POSIX-IO interface as the storage layer. Typical examples of such file systems used on most HPC platforms are Lustre [21] and OrangeFS [22].

Beyond the POSIX-IO interface lies the POSIX-IO semantics that a fully compliant file systems must implement. This standard has advantages regarding portability, but its inflexibility can cause considerable performance degradation [23]. For example, this standard requires that changes made to a shared file must be visible immediately by all processes. Because an application has no way of telling the file system that POSIX-IO semantics are unnecessary or unwanted, it cannot avoid this performance penalty. For many file systems, these performance issues are noticeable even for small numbers of client processes and straightforward I/O patterns [24, 25]. The issues also affect higher levels of the I/O stack because an underlying POSIX-compliant file system effectively forces POSIX-IO semantics upon all other layers . For instance, this applies to the common HPC I/O stack with Lustre [26].

2.2. Big data: from file systems to object storage

Big data applications require a storage model that follows a write-once, read-many model. This requirement drove the design of many distributed file systems by sacrificing some of the POSIX-IO operations in order to gain data throughput. GFS [27] implements only a set POSIX-compliant operations needed by data-intensive applications, namely, create, delete, open, close, read, and write. HDFS [11] is based on GFS and is designed to work in commodity hardware. It implements some additional POSIX-IO requirements such as directory operations and file permissions, but it discards some others such as concurrent reads and writes. Ceph [4] follows the same trend, discarding some POSIX-IO semantics and implementing only those that allow a distributed file system to work with most applications. GlusterFS eliminates the metadata server and claims to be fully POSIX compliant. However, work has shown that this compliance can impact throughput [28].

We can clearly see a trend where the file system POSIX-IO API or semantics such as providing a hierarchical namespace, file permissions, or strict file access parallelism are unnecessary. Thus, they can be traded for performance and adaptability for Big Data.

2.3. Storage convergence between HPC and Big Data

During the past decade many research projects and workshops were dedicated to the opportunities and possibilities of running large scale scientific applications by using cloud computing technologies [29, 30]. In general, many efforts there were made to investigate the performance of HPC applications (mostly form life sciences [31]) on clouds (with and without virtualization [32]) highlighting cost efficiency or trade-offs [33].

Several research efforts focus on building optimized or customized distributed-computing platforms that meet the requirements of HPC applications and scientific simulations [34]. Many of those are based on big data frameworks such as Spark or Hadoop / MapReduce. In contrast, Pan et al. [35] propose to port parallel file systems to cloud environments in order to support a wide range of applications expecting POSIX-IO on cloud applications. However, the application use cases considered in that work are rarely data-intensive. In the same way other researchers also aimed to provide the features of PFS in the cloud storage. For instance, Y. Abe and G. Gibson in [36] presented a storage model which gives data access to a user through the storage service layer (S3 interface) and directly through a PFS.

3. Could blobs be the enabling factor?

Although the set of tools and techniques used for HPC and big data environment differ, many objectives are similar. The most important is probably to provide the highest-possible data access performance and parallelism. As such, the storage stack for HPC and Big Data looks similar. Indeed, the related work showed that a common trend for both HPC and big data is to relax many of the concurrent file access semantics, trading such strong guarantees for increased performance. Nevertheless, some differences remain. Specifically, while the big data community increasingly drops POSIX-IO altogether, the HPC community tends to provide this relaxed set of semantics behind the same API. Although this choice increases backwards compatibility with legacy applications, it also has significant performance impact.

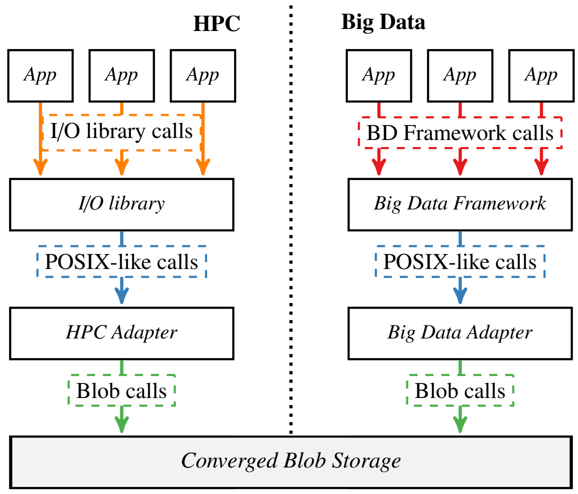

Figure 1: Converged side-to-side storage stack for HPC and Big Data.

Very few HPC applications actually rely on strong POSIX-IO semantics. For instance, the MPI-IO standard does not only relaxes many of these semantics but also drops many of its operations altogether. For example, it does not expose the file hierarchy or permissions to the end user. Therefore, applications leveraging these libraries do not need these features to be provided.

Consequently, we ask ourselves whether the underlying storage technologies from both worlds could be unified, leading to specific software stacks and libraries running atop a low-level, low-opinionated storage paradigm, such as the one provided by blob storage systems (Figure 1). Indeed, blob-based storage systems such as Týr [20] or Rados [18] could provide a strong alternative to file-based storage on both sides. These systems typically have a much more limited set of primitive compared with file systems:

- **Blob Access**: object read, object size,
- **Blob Manipulation**: object write, truncate,
- **Blob Administration**: create object, delete object,
- **Namespace Access**: scan all objects.

These operations are similar to those permitted by the POSIX-IO API on a single file. Therefore, most file operations performed on a file system can be mapped directly to the corresponding primitives of blob storage systems. In that model we classify file open and unlink as file operations.

In contrast, directory-level operations do not have their blob counterpart, because of the flat nature of the blob namespace. Should applications need them, such operations can be emulated using the scan operation.

Table 1: Application summary

Platform	Application	Usage	Total reads	Total writes	R / W ratio	Profile
HPC / MPI	mpiBLAST (BLAST)	Protein docking	27.7 GB	12.8 MB	2.1×10^{3}	Read-intensive
	MOM	Oceanic model	19.5 GB	3.2 GB	6.01	Read-intensive
	ECOHAM (EH)	Sediment propagation	0.4 GB	9.7 GB	4.2×10^{-2}	Write-intensive
	Ray Tracing (RT)	Video processing	67.4 GB	71.2 GB	0.94	Balanced
Cloud / Spark	Sort	Text Processing	5.8 GB	5.8 GB	1.00	Balanced
	Connected Component (CC)	Graph Processing	13.1 GB	71.2 MB	0.18	Read-intensive
	Grep	Text Processing	55.8 GB	863.8 MB	64.52	Read-intensive
	Decision Tree (DT)	Machine Learning	59.1 GB	4.7 GB	12.58	Read-intensive
	Tokenizer	Text Processing	55.8 GB	235.7 GB	0.24	Write-intensive

Obviously, this emulation is far from optimized. Yet, since we expect these calls to be vastly outnumbered by blob-level operations, this performance drop is likely to be compensated by the gains permitted by using a flat namespace and simpler semantics.

While some of these features (*e.g.*, permissions) are usually not used at the application level, they may be necessary at the system management level, for multi-user systems as provided on HPC environments. In the context of blob-based storage systems, we advocate that such access control can be provided using multiple keyspaces (*e.g.*, per-application keyspaces), in which these features can be configured at the keyspace level.

Legacy application, which could rely on a fully compliant POSIX-IO interface, could leverage a POSIX-IO interface implemented atop such blob storage. This is proven possible by the Ceph file system, a file-system interface to Rados.

4. A representative set of applications

The challenges posed by convergence between HPC and Big Data applications have raised many discussions in the community [29]. One of the emerging ideas from these discussions is that the applications cannot be considered separately from the underlying software stack; the fuel for convergence could enable a wide variety of HPC and Big Data applications to leverage converging services and underlying infrastructure. Although designing a full converging stack is not the objective of this paper, we focus on storage as one of the critical milestones that could make such convergence possible. Specifically, we choose I/O-intensive applications, which could benefit most from such converged storage.

Accordingly, we base our experiments on a number of I/O-intensive applications extracted from the literature [37, 38] that cover the diversity of I/O workloads commonly encountered on both HPC (Section 4.1) and Big Data platforms (Section 4.2).

4.1. HPC Applications

The HPC applications we use are based on MPI. They all leverage either large input of output datasets associated with large-scale computation atop centralized storage usually provided by a distributed, POSIX-IO-compliant file system such as Lustre [21].

mpiBLAST [39] is a parallel MPI implementation of NCBI BLAST [40]. It is a read-intensive biomolecular tool searching for regions of similarity between biological sequences. It compares nucleotide or protein sequences to sequence databases and calculates the statistical significance.

MOM (Modular Ocean Model) [41] is a read-intensive three-dimensional ocean circulation model targeted at understanding the ocean climate system.

ECOHAM5 (ECOlogical model, HAMburg, version 5) [42] is a write-intensive three-dimensional biogeochemical ecosystem model with the focus on the North Sea [43, 44]. It modelizes the pelagic and benthic cycles of carbon, nitrogen, phosphorus, silicon and oxygen on the northwest European continental shelf.

Ray Tracing is a balanced read-write workload extracted from the BigDataBench [38], itself derived from [45]. It generates images by tracing the path of light and simulating the effects of its encounters with virtual objects.

4.2. Big Data Applications

As the leading open-source Big Data processing and analytics framework, Apache Spark [46] appears as an ideal candidate for this research. Chosen applications are extracted from SparkBench [47], a benchmarking suite for Spark. It comprises a representative set of workloads belonging to four application types: machine learning, graph processing, streaming, and SQL queries.

Sort is a widely used benchmark that reads input data and sorts it based on a given key. It is I/O-intensive since all the data read will be processed and written back to the file system. For example, it can be used to sort a series of readings from sensors by date.

Grep is a filtering workload that searches in the input data for lines containing a given word and saves these lines into HDFS. In contrast to Sort, the size of the input and output will not be equal, and some data will be filtered out.

Decision Tree is a machine learning workload that reads a dataset containing rows with a series of features and a class they belong to. This dataset is then split into a training and a test set. The workload creates a predictive model with the training set that is able to predict the class of the elements in the test set. These predictions are written back to disk.

Connected Component is an algorithm that finds the subgraphs in a graph in which any two vertices are connected by a path but are not connected to any other node on the supergraph. This can be seen as a way of finding clusters of nodes. For example, in a social network it can be used to find communities of users. The implementation in the benchmark will read a dataset and write the labels of each component back to disk.

Tokenizer is a Spark application we developed that reads a text file, tokenizes each line into words, and for each line calculates the `NGrams=2` (i.e., contiguous sequences of 2 words from each line). These Ngrams are saved in a text file. This is a common preprocessing step in topic modeling for documents where word patterns have to be extracted as an input to a machine learning model. This application shows a write-intensive workload.

5. Experimental configuration

We run experiments using the Grid'5000 [48] experimental testbed, which spans 11 sites in France and Luxembourg. In this paper we use 32 nodes of the *parapluie* cluster in Rennes. Each node embeds 2 x 12-core 1.7 Ghz 6164 HE, 48 GB of RAM, and 250 GB HDD. We use Gigabit Ethernet connectivity (MTU = 1500 B) for Big Data applications, and 4 x 20G DDR Infiniband for HPC applications. HPC applications ran atop Lustre 2.9.0 and MPICH 3.2, using multiple ratios of storage-to-compute nodes. Big Data application ran atop Spark 2.1.0, Hadoop / HDFS 2.7.3 and Ceph Kraken.

All storage systems are configured with similar parameters to allow for a fair comparison. Specifically, each system is configured with a replication factor of 1 (no replication), and using a stripe size of 64 MB.

In addition, we prove that the results obtained with HPC applications are replicable to a high-end supercomputer. To do so, we performed extra experiments on the Theta supercomputer [15] hosted at the Argonne Leadership Computing Facility (ALCF). Theta is a last-generation 9.65-petaflop Cray XC40 system. It is composed of 3,624 nodes, each containing a 64 core Intel Knights Landing processor with 16 GB of high-bandwidth in-package memory (MCDRAM), an additional 192 GB of DDR4 RAM, and a 128 GB local SSD.

6. Analyzing the distribution of storage calls

In this section we demonstrate that the actual I/O calls made by both HPC and big data applications are not incompatible with the set of features provided by state-of-the-art blob storage systems. Our intuition is that read and write calls are vastly predominant in the workloads of those applications and that other features of distributed file systems such as directory listings are rarely used, if at all.

6.1. Tracing HPC applications

Figure 2 summarizes the relative count of storage calls performed by our set of HPC applications. The most important observation for all four applications is the predominance of reads and writes. Except for ECOHAM, no application performed any other call to the storage system that reads or writes files, thus confirming our first intuition. This was expected because the MPI-IO standard does not permit any other operation.

The few storage calls other than read and write (mainly extended attributes reads and directory listings) are due to the run script necessary to prepare the run and collect results after it finishes. These steps can be performed offline from the I/O-heavy MPI part of the application. This results in only reads and writes being performed (EH / MPI).

We conclude that the only operations performed by our set of HPC applications, namely, file I/O, can be mapped to blob I/O on a blob storage system. Consequently, these applications appear to be suited to run unmodified atop blob storage.

6.2. Tracing Big Data applications

Figure 3 shows the relative count of storage calls performed by our set of big data applications to HDFS.

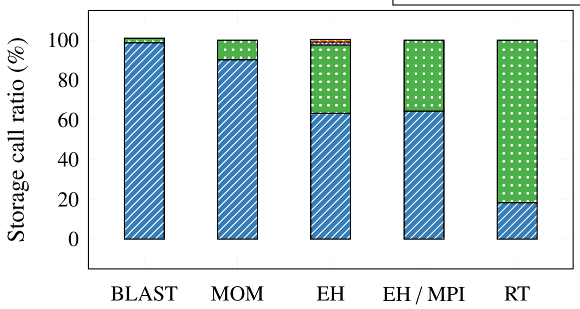

Figure 2: Measured relative amount of different storage calls to the persistent file system for HPC applications

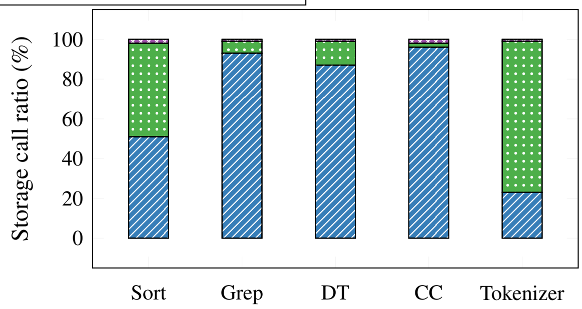

Figure 3: Measured relative amount of different storage calls to the persistent file system for Big Data applications

Table 2: Spark directory operation breakdown

Operation	Action	Count
`mkdir`	Create directory	43
`rmdir`	Remove directory	43
`opendir` *(Input data directory)*	Open / List directory	5
`opendir` *(Other directories)*	Open / List directory	0

Similar to what we observed with HPC applications, the storage calls are vastly dominated by reads and writes to files. In contrast with HPC, however, all applications also cause Spark to perform a handful of directory operations (86 in total across all our applications). These directory operations are not related to the data processing because input / output files are accessed directly by using read and write calls.

Analyzing these directory operations, we notice that they are related solely to (*a*) creating the directories necessary to maintain the logs of the application execution, (*b*) listing the input files before each application runs if the input data is set as a directory, and (*c*) maintaining the `.sparkStaging` directory. This directory is internally used by Spark to share information related to the application between nodes and is filled during the application submission. It contains application files such as the Spark jar or the application jar, as well as distributed cache files [49].

We analyze in detail the directory operations performed by big data applications. Table 2 shows the breakdown of all such directory operations across all applications by storage call. We note that only the input data directories are listed, meaning that Spark accesses directly all the other files it needs with their path. Consequently, a flat namespace such as the one provided by blob storage systems could probably be used.

7. Big Data: Hierarchical to flat namespace

We previously observed that Spark storage calls included several very rare directory operations. Our observations concluded that such calls are only performed for separating data files from temporary files and are not strictly necessary for the application itself. Yet, one of our objectives is to prove that Spark applications can run *unmodified* atop blob storage. We prove that Spark can run atop a flat namespace providing only object-level operations, as provided by blob storage systems; such a proof would confirm our previous assertions.

Since file operations dominate directory operations, we choose to optimize the former at the expense of the latter. Thus, for any given hierarchical file path we generate a predictable flat path. Subsequent operations to that path are translated to a file operation on the rewritten path. In HDFS we achieve this by storing all files at the root. That is, we store on path `/foo__bar` a file that would normally be stored on `/foo/bar` in a hierarchical namespace. Listing or deleting a directory is implemented by scanning and filtering the whole set of files in the system, selecting only the matching files that would be contained in that folder. Although such scan operations are costly, they are infrequent. Consequently the performance gains by flattening the file system (i.e., not managing permissions) should outweigh the cost of such operations. Table 3 summarizes the rewrite rules we apply on incoming storage calls from Spark.

We intercept and rewrite storage calls by modifying Hadoop / HDFS file storage interface. We run all Big Data applications 100 times and compare the average completion time of each benchmark suite on HDFS us-

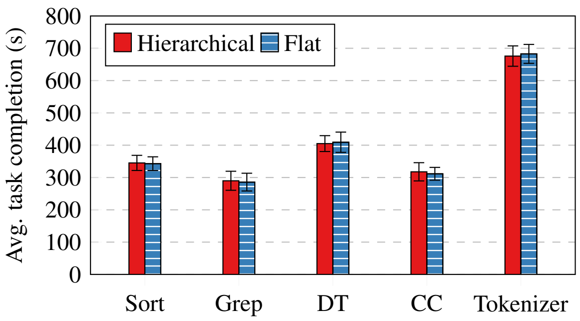

Figure 4: Average Big Data task completion time with and without flat namespace simulation, with 95% confidence intervals.

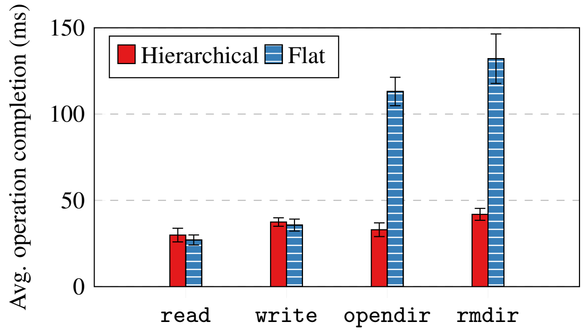

Figure 5: Average individual Big Data operation completion time with and without flat namespace simulation, with 95% confidence intervals.

Table 3: Big Data storage call translation rules

Original operation	Rewritten operation
`create(/foo/bar)`	`create(/foo__bar)`
`open(/foo/bar)`	`open(/foo__bar)`
`read(fd)`	`read(bd)`
`write(fd)`	`write(bd)`
`mkdir(/foo)`	*Dropped operation*
`opendir(/foo)`	`scan(/)`, *return all files matching* `/foo__*`
`rmdir(/foo)`	`scan(/)`, *remove all files matching* `/foo__*`

ing the original hierarchical namespace or simulating a flat namespace. We plot the results in Figure 4. Flattening the namespace on HDFS does not result in any significant task completion time variation despite the higher completion time of directory operations, plotted in Figure 5. Indeed, these calls are diluted in the vastly superior amount of file read and write calls.

We demonstrated that running our set of Big Data applications over a simulated flat namespace not only is possible but also does not cause any significant performance variation. Consequently, these applications also appear to be suited to run atop a blob storage system, thus further enhancing the performance of the application by leveraging the reduced complexity of managing a flat namespace.

8. Replacing file-based by blob-based storage

In this section we demonstrate the potential of blob-based storage to suit the storage needs of both HPC and Big Data applications. We deploy each application listed in Section 4 atop state-of-the-art blob storage systems. We detail these systems in Section 8.1. We prove that the performance of these applications running atop converged blob-based storage matches or exceeds that of the same applications running atop Lustre for HPC (Section 8.2) and HDFS for Big Data (Section 8.4).

8.1. Overview of the blob storage systems

We run our applications atop two state-of-the-art blob storage systems: Týr [20] and Rados [18]. We are using only the basic blob storage functionality they provide, and do not make use of any advanced features they may support. Although their high-level design has similarities, these two systems have different strengths and weaknesses resulting from design decisions made for each to support specific use cases.

Týr is a large-scale blob storage system designed around the same design principles as the Dynamo key-value database [50]. It is targeted at high access parallelism using multiversion concurrency control (MVCC) associated with built-in multiobject transactions. Týr offers fine-grained random write access to data, as well as single-hop reads (i.e., accessing the storage server without prior communication with any metadata server).

Rados is the storage layer for Ceph FS [4]. Rados has the ability to scale to thousands of hardware devices by using management software that runs on each of the individual storage nodes. The software provides features such as thin provisioning, snapshots and replication.

We assess the performance impact of replacing file-based with blob-based storage by observing three metrics. The *job completion time* is the total execution time of the application, from submission to completion. *Read bandwidth* and *write bandwidth* respectively represent the average data transferred per unit of time for read and write requests. We collect all these metrics on

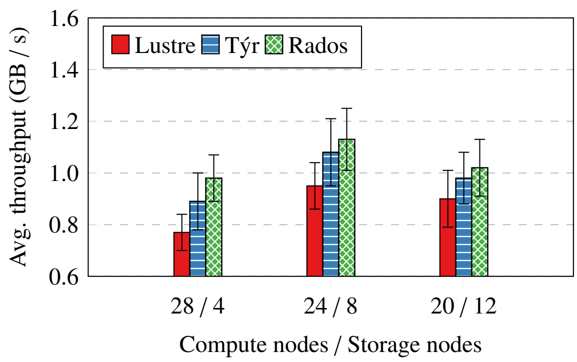

Figure 6: Average agreggate throughput across all HPC applications varying the compute-to-storage ratio, with 95% confidence intervals.

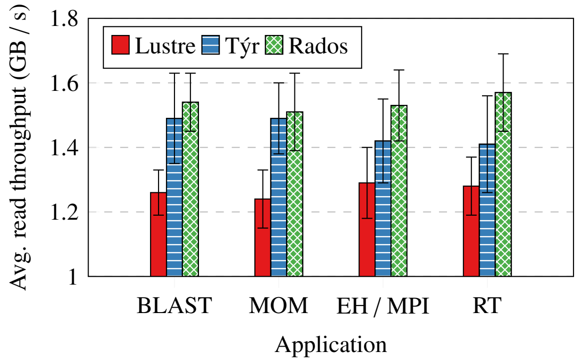

Figure 7: Comparison of read throughput for each HPC application with Lustre, Týr and Rados, with 95% confidence intervals.

the compute nodes by instrumenting the adapter, and we aggregate the results.

8.2. Replacing Lustre with blob-based storage on HPC

In this section, we demonstrate how blob-based storage system can be used to support HPC applications while matching or exceeding Lustre I/O performance by replacing the latter with Týr and Rados.

We experiment using three storage-to-compute node configurations in order to ensure that our results are independent of the cluster configuration. We run the same experiments respectively with 28 compute / 4 storage nodes, 24 / 8 and 20 / 12. We average the results of 100 experiment runs.

On each node, we deploy a small interceptor to redirect POSIX storage calls to the blob storage system. It is based on FUSE [51], which is supported on most Linux kernels today. In that configuration, this interceptor acts as the HPC adapter as presented in Figure 1. This adapter translates file operations to blob operations according to Table 4. Directory operations are not supported as we showed previously that they are unnecessary for HPC applications. The APIs of the blob storage systems we consider allow for a direct mapping between file-based and blob-based storage operations. We partially implement the `stat` function. Specifically, the file size is mapped to the blob size, the permissions are set to 777, the block size and allocated block size are set to 512 bytes, and the inode number is set to the hash of the blob key. The remaining information is set to 0. Our implementation does not support symbolic or hard links, which are not needed by our applications.

In Figure 6 we plot the average aggregate read and write bandwidth for all applications while varying the compute-to-storage node ratio. We note that for our configuration the 24 compute node / 8 storage node

Table 4: HPC storage call translation rules

POSIX Call	Translated Call
`create(/foo/bar)`	`create(/foo__bar)`
`open(/foo/bar)`	`open(/foo__bar)`
`read(fd)`	`read(bd)`
`write(fd)`	`write(bd)`
`mkdir(/foo)`	*Unsupported operation*
`opendir(/foo)`	*Unsupported operation*
`rmdir(/foo)`	*Unsupported operation*

setup results in the higher bandwidth for all storage systems. Hence, the following experiments are performed with that configuration. This ratio is much lower than on common HPC platforms (3:1 vs. ∼ 70:1 at ORNL, for instance [52]) mainly because the jobs we run are significantly more data-intensive than compute-intensive. We note from these results that blob storage systems constantly outperform Lustre in all configurations for both reads and writes. We will detail these results in the following experiments. For read-intensive applications such as BLAST and MOM, this performance increase allows blob storage systems with 4 storage nodes to achieve a bandwidth comparable to Lustre's with 8 storage nodes.

In Figure 7 we plot the average read bandwidth for each of our HPC applications with Lustre file-based storage and Týr or Rados blob-based storage. We note an average 14% reduction of the total read time when using blob-based storage compared to Lustre. This is because of the optimized write path of the two blob storage systems considered. Indeed, both enable clients to locate and access any piece of data directly without prior communication with any dedicated metadata node. Although both blob storage systems behave similarly with respect to read performance, Rados shows a higher read performance due to its lower read consistency.

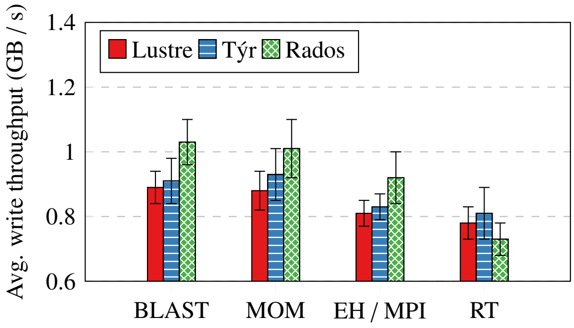

Figure 8: Comparison of write throughput for each HPC application with Lustre, Týr, and Rados, with 95% confidence.

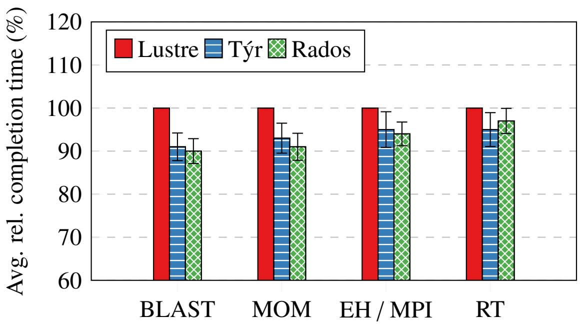

Figure 9: Average performance improvement relative to Lustre for HPC applications using blob-based storage, with 95% confidence.

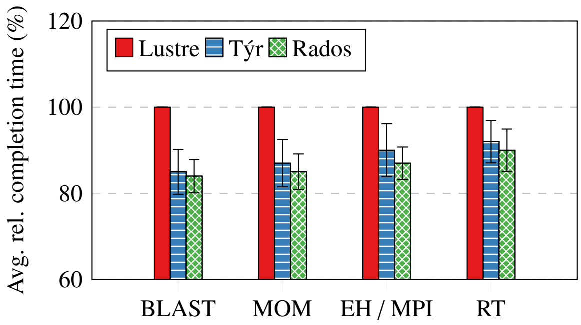

Figure 10: Average performance improvement relative to Lustre for HPC applications using blob storage, with 95% confidence on Theta

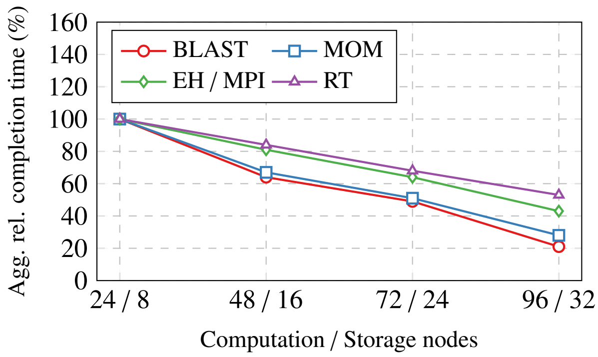

Figure 11: Average performance improvement at scale relative to 32 nodes setup for HPC applications using blob-based storage on Theta.

We plot in Figure 8 the average write bandwidth obtained in similar conditions. These results show that blob storage systems outperform Lustre for BLAST, MOM and ECOHAM. Interestingly, for Ray Tracing, Rados performace drops behind that of other systems. This is the result of the relatively small writes (8 KB on average) performed by this application compared to the other systems, causing high lock contention on concurrent writes that hinders the throughput of the system.

In Figure 9 we plot the average application completion time improvement. The I/O performance gains are here diluted in compute operations. As expected considering the previous results, read-intensive applications exhibit the greatest decrease. BLAST and MOM show a completion time reduction of nearly 8% with both blob storage systems. In contrast, write-intensive applications such as ray tracing show a lower 3% completion time decrease with Týr or Rados as the underlying storage when compared with Lustre.

8.3. Replicating results on a high-end supercomputer

In this section, we seek to prove that the experiments obtained in Section 8.2 are reproducible on a high-end supercomputer. To do so, we leverage the Theta supercomputer hosted at Argonne National Laboratory, and run the same experiments as in the aforementioned section. Although arguably not an easy task due to the strong limitations of the platform and the lack of superuser rights, blob storage systems such as Týr and Rados are deployable on such platform.

We deploy the applications as described in the previous section, using 32 nodes (totaling 2048 cores), using 24 nodes for computation and 8 nodes for storage, and measure the completion time for each application. Experiments with Lustre were using the file system available to the computer, totaling 170 storage nodes and shared across all users.

We plot the results in Figure 10. We show the performance improvement to be significantly higher than on our testbed. The reason for this significant perfor-

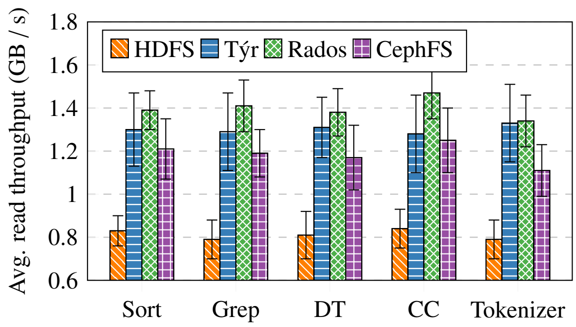

Figure 12: Comparison of read throughput for each Big Data application with HDFS, Týr, Rados and CephFS.

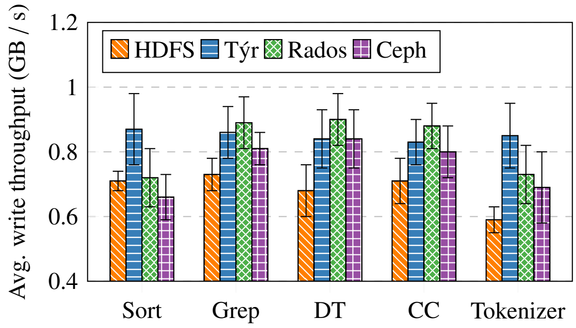

Figure 13: Comparison of write throughput for each Big Data application with HDFS, Týr, Rados and CephFS.

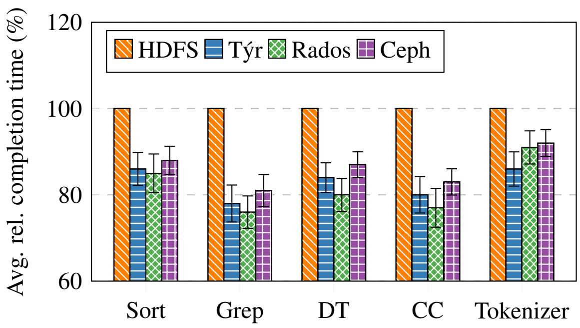

Figure 14: Average performance improvement relative to HDFS for Big Data applications using blob-based storage.

mance increase is to be found in the technical characteristics of Theta, which offers significantly more RAM than SSD space. As such, for the most part, the storage systems deployed on these nodes behave as in-memory storage systems. We acknowledge that the setup of this platform is particular in this regard and by consequence that the results are not representative of those of another platform with different setup. Yet, we advocate that the results reach the goal of demonstrating that deploying blob storage systems on high-end HPC platforms is possible without requiring any application modification. We observe a higher variance in the results compared to Grid'5000, which we attribute to the shared nature of the centralized storage.

In Figure 11 we scale all four applications on up to 128 nodes of Theta, or 8192 cores. Because of allocation limitations, each experiment was performed only 5 times. We notice a near-linear decrease of computation time as the size of the cluster increases. With applications applications such as ECOHAM or Ray Tracing, the performance improvement is slightly lower than with purely read-intensive applications due to the higher cost of write operations compared to read operations.

8.4. Running Spark applications atop blob storage

In this section we run the same set of experiments for the set of Big Data applications. We demonstrate that Týr and Rados significantly outperform HDFS for all applications. In order to provide an additional baseline of the performance of file systems, we also run these applications atop CephFS [4], itself based on Rados.

We use the same configuration as in Section 7, running computation alongside storage on 32 nodes. We integrate the storage adapter for blob storage directly inside HDFS. The Hadoop installation has been modified to redirect storage calls to blob storage systems. The translation between POSIX-like calls and flat-namespace blob operations is done by using the translation rules defined in Table 3. We implement the CRUSH algorithm to provide Spark with the physical data location on Rados and CephFS. We use the client API capabilities to provide that information with Týr.

In Figure 12 we plot the average read bandwidth achieved for each of the Spark benchmarks. We notice a striking read bandwidth improvement when using Týr and Rados over HDFS (the read bandwidth is increased by 28% and 32% respectively). Same as in HPC, this is due mainly to the direct read feature of the two storage systems that, unlike HDFS, enable the applications to bypass any centralized metadata service and access the storage servers directly. CephFS performance highlights the cost of file-based storage by showing a degraded performance compared to Rados. As with HDFS, this is mostly due to the additional communication required with dedicated metadata servers in the critical path for read requests, made necessary by the file hierarchy management.

Figure 13 shows the average write bandwidth for these applications. Similar to what was observed with

HPC, we note a constant write bandwidth improvement with blob storage over HDFS and CephFS. We also note the same pattern we observed with HPC. Specifically, Rados outperforms Týr on read-intensive applications, whereas Týr enables higher throughput on write-intensive applications. This is visible with the Tokenizer application, where lock contention due to lack of multi-version concurrency control in Rados causes significant performance loss on concurrent write access.

In Figure 14 we plot the relative improvement in the total application completion time, diluted in computation. Running Big Data applications atop blobs improves application cpmpletion time, up to 22% compared to HDFS and 7% compared to CephFS. For Big Data, the highest gains are obtained with read-intensive applications such as Grep and Decision Tree. In comparison, write-intensive applications such as Tokenizer also benefit from improved performance, although relatively smaller due to the globally greater complexity of the write protocols for each storage system.

9. Conclusion

In this paper we argue that blob storage is a strong candidate for replacing traditional storage for both HPC and Big Data. Its simple data model is enough to map directly file operations to blob operations. Based on the previous observation that simple file reads and writes constitute the vast majority of the storage calls made by both HPC and Big Data applications, we factually prove that this convergence is possible by mapping both HPC and Big Data applications to blob storage. This does not require any modification in the application thanks to a thin adapter layer between the application and the persistent storage. We leverage 4 real-life HPC applications as well as 5 Big Data benchmarks to prove on an experimental testbed that not only such convergence is possible, but that it also significantly improve performance by up to 25% with read-intensive applications. We confirm on the Theta supercomputer that this setup is reasonable and applicable to a high-end supercomputer with near-linear scalability up to 8,192 cores.

In future work we will experiment both Týr and Rados at the same time on a real supercomputer as well as on one of the leading cloud computing platforms, both with a larger set of applications and frameworks. We will compare our approach with an extensive set of competitor storage systems and platform configurations.

Acknowledgments

This work is part of the "BigStorage: Storage-based Convergence between HPC and Cloud to handle Big Data" project, H2020-MSCA-ITN-2014-642963, funded by the European Commission within the Marie Skłodowska-Curie Actions framework. This work was supported by the U.S. Department of Energy, Office of Science, Advanced Scientific Computing Research under Contract DE-AC02-06CH11357. It is also supported by the ANR OverFlow project, ANR-15-CE25-0003. Experiments presented in this paper were carried out using the Grid'5000 testbed, supported by a scientific interest group hosted by Inria and including CNRS, RENATER, and several universities and organizations.

References

[1] Apache Cassandra, `https://cassandra.apache.org/`.

[2] Project Voldemort, `http://www.project-voldemort.com/voldemort/`.

[3] Openstack Swift, `https://docs.openstack.org/swift/latest/`.

[4] S. A. Weil, S. A. Brandt, E. L. Miller, et al., Ceph: A scalable, high-performance distributed file system, in: Proceedings of the 7th Symposium on Operating Systems Design and Implementation, OSDI '06, USENIX Association, Berkeley, CA, USA, 2006, pp. 307–320.

[5] N. Liu, J. Cope, et al., On the role of burst buffers in leadership-class storage systems, in: IEEE 28th Symposium on Mass Storage Systems and Technologies, MSST 2012, April 16-20, 2012, Asilomar Conference Grounds, Pacific Grove, CA, USA, 2012, pp. 1–11.

[6] M. Romanus, R. B. Ross, M. Parashar, Challenges and considerations for utilizing burst buffers in high-performance computing, CoRR abs/1509.05492.

[7] Q. Liu, J. Logan, Y. Tian, et al., Hello ADIOS: the challenges and lessons of developing leadership class I/O frameworks, Concurrency and Computation: Practice and Experience 26 (7) (2014) 1453–1473.

[8] C. Bartz, K. Chasapis, M. Kuhn, P. Nerge, T. Ludwig, A best practice analysis of HDF5 and NetCDF-4 using lustre, in: High Performance Computing, no. 9137 in Lecture Notes in Computer Science, Springer International Publishing, Switzerland, 2015, pp. 274–281.

[9] M. P. Forum, Mpi: A message-passing interface standard, Tech. rep., Knoxville, TN, USA (1994).

[10] M. Zaharia, R. S. Xin, P. Wendell, et al., Apache spark: A unified engine for big data processing, Commun. ACM 59 (11) (2016) 56–65.

[11] K. Shvachko, H. Kuang, S. Radia, et al., The Hadoop distributed file system, in: 2010 IEEE 26th symposium on Mass Storage Systems and Technologies (MSST), IEEE, 2010, pp. 1–10.

[12] Microsoft Azure, `https://azure.microsoft.com/en-us/`.

[13] Amazon Web Services, `https://aws.amazon.com/`.

[14] MareNostrum, `https://www.bsc.es/marenostrum/marenostrum`.

[15] Theta, `https://www.alcf.anl.gov/theta`.

[16] Q. Zheng, K. Ren, et al., Deltafs: Exascale file systems scale better without dedicated servers, in: Proceedings of the 10th Parallel Data Storage Workshop, PDSW '15, 2015, pp. 1–6.

[17] P. Carns, J. Jenkins, C. D. Cranor, et al., Enabling NVM for data-intensive scientific services, in: 4th Workshop on Interactions of NVM/Flash with Operating Systems and Workloads (INFLOW 16), USENIX Association, GA, 2016.
[18] S. A. Weil, A. W. Leung, S. A. Brandt, C. Maltzahn, Rados: A scalable, reliable storage service for petabyte-scale storage clusters, in: Proceedings of the 2nd international workshop on petascale data storage: held in conjunction with Supercomputing'07, ACM, 2007, pp. 35–44.
[19] B. Nicolae, G. Antoniu, L. Bougé, et al., Blobseer: Next-generation data management for large scale infrastructures, J. Parallel Distrib. Comput. 71 (2) (2011) 169–184.
[20] P. Matri, A. Costan, G. Antoniu, J. Montes, M. S. Pérez, Týr: Blob storage meets built-in transactions, in: SC16: International Conference for High Performance Computing, Networking, Storage and Analysis, 2016, pp. 573–584.
[21] The Lustre File System, `http://lustre.org/` (2017).
[22] M. Moore, D. Bonnie, W. Ligon, et al., OrangeFS: Advancing PVFS, in: 2011 9th USENIX Conference on File and Storage Technologies (FAST), 2011.
[23] D. Kimpe, R. Ross, Storage models: Past, present, and future, High Performance Parallel I/O (2014) 335–345.
[24] J. Cope, K. Iskra, et al., Bridging HPC and grid file I/O with IOFSL, in: Applied Parallel and Scientific Computing - 10th International Conference, PARA 2010, Reykjavík, Iceland, June 6-9, 2010, Revised Selected Papers, Part II, 2010, pp. 215–225.
[25] D. Huang, J. Yin, et al., UNIO: A unified I/O system framework for hybrid scientific workflow, in: Cloud Computing and Big Data - Second International Conference, CloudCom-Asia 2015, Huangshan, China, June 17-19, 2015, Revised Selected Papers, 2015, pp. 99–114.
[26] M. Kuhn, A Semantics-Aware I/O Interface for High Performance Computing, in: Supercomputing, no. 7905 in Lecture Notes in Computer Science, Springer, Berlin, Heidelberg, 2013, pp. 408–421.
[27] S. Ghemawat, H. Gobioff, S.-T. Leung, The Google file system, in: ACM SIGOPS Operating Systems Review, Vol. 37, ACM, 2003, pp. 29–43.
[28] S. Mikami, K. Ohta, O. Tatebe, Using the gfarm file system as a POSIX compatible storage platform for Hadoop MapReduce applications, in: Proceedings of the 2011 IEEE/ACM 12th International Conference on Grid Computing, IEEE Computer Society, 2011, pp. 181–189.
[29] BDEC – Big Data and Extreme-Scale Computing, `http://www.exascale.org/bdec/` (2017).
[30] Z. Zhang, K. Barbary, F. A. Nothaft, et al., Scientific computing meets big data technology: An astronomy use case, in: 2015 IEEE International Conference on Big Data, Big Data 2015, Santa Clara, CA, USA, October 29 - November 1, 2015, 2015, pp. 918–927.
[31] H. A. Duran-Limon, J. Flores-Contreras, et al., Efficient execution of the WRF model and other HPC applications in the cloud, Earth Science Informatics 9 (3) (2016) 365–382.
[32] A. Jaikar, S. Noh, Cloud computing: Read before use, T. Large-Scale Data- and Knowledge-Centered Systems 30 (2016) 1–22.
[33] A. Gupta, D. Milojicic, Evaluation of HPC applications on cloud, in: Open Cirrus Summit (OCS), 2011 Sixth, IEEE, 2011, pp. 22–26.
[34] R. Ledyayev, H. Richter, High performance computing in a cloud using OpenStack, Cloud Computing (2014) 108–113.
[35] A. Pan, J. P. Walters, V. S. Pai, et al., Integrating high performance file systems in a cloud computing environment, in: 2012 SC Companion: High Performance Computing, Networking Storage and Analysis, 2012, pp. 753–759.
[36] Y. Abe, G. Gibson, pwalrus: Towards better integration of parallel file systems into cloud storage, in: Cluster Computing Workshops and Posters (Cluster Workshops), 2010 IEEE International Conference on, IEEE, 2010, pp. 1–7.
[37] G. C. Fox, J. Shantenu, et al., Towards a comprehensive set of big data benchmarks, Advances in Parallel Computing 26 (2015) 47–66.
[38] L. Wang, J. Zhan, C. Luo, et al., BigDataBench: A big data benchmark suite from Internet services, in: 2014 IEEE 20th International Symposium on High Performance Computer Architecture (HPCA), IEEE, 2014.
[39] mpiBLAST: Open-Source Parallel BLAST, `http://www.mpiblast.org/` (2017).
[40] S. F. Altschul, W. Gish, et al., Basic local alignment search tool, Journal of Molecular Biology 215 (3) (1990) 403–410.
[41] Ocean circulation models, `https://www.gfdl.noaa.gov/ocean-model/` (2017).
[42] U. H. Institute of Oceanography, ECOHAM, `https://wiki.zmaw.de/ifm/ECOHAM` (2015).
[43] I. Lorkowski, J. Pätsch, A. Moll, W. Kühn, Interannual variability of carbon fluxes in the North Sea from 1970 to 2006–competing effects of abiotic and biotic drivers on the gas-exchange of CO_2, Estuarine, Coastal and Shelf Science 100 (2012) 38–57.
[44] F. Große, N. Greenwood, et al., Looking beyond stratification: A model-based analysis of the biological drivers of oxygen depletion in the North Sea, Biogeosciences Discussions (2015) 2511–2535.
[45] J. Stone, An efficient library for parallel ray tracing and animation, Tech. rep., Intel Supercomputer Users Group Proceedings (1995).
[46] M. Zaharia, M. Chowdhury, T. Das, et al., Resilient distributed datasets: A fault-tolerant abstraction for in-memory cluster computing, in: Proceedings of the 9th USENIX conference on Networked Systems Design and Implementation, USENIX Association, 2012, pp. 2–2.
[47] M. Li, J. Tan, Y. Wang, L. Zhang, V. Salapura, Sparkbench: a comprehensive benchmarking suite for in memory data analytic platform Spark, in: Proceedings of the 12th ACM International Conference on Computing Frontiers, ACM, 2015, p. 53.
[48] D. Balouek, A. Carpen Amarie, et al., Adding virtualization capabilities to the Grid'5000 testbed, in: Cloud Computing and Services Science, Vol. 367 of Communications in Computer and Information Science, Springer International Publishing, 2013, pp. 3–20.
[49] Running Spark on YARN, `https://spark.apache.org/docs/latest/running-on-yarn.html` (2017).
[50] G. DeCandia, D. Hastorun, M. Jampani, et al., Dynamo: Amazon's highly available key-value store, SIGOPS Oper. Syst. Rev. 41 (6) (2007) 205–220.
[51] S. Ishiguro, J. Murakami, et al., Optimizing local file accesses for fuse-based distributed storage, in: 2012 SC Companion: High Performance Computing, Networking Storage and Analysis, Salt Lake City, UT, USA, November 10-16, 2012, 2012, pp. 760–765.
[52] S. Oral, J. Simmons, J. Hill, et al., Best practices and lessons learned from deploying and operating large-scale data-centric parallel file systems, in: SC14: International Conference for High Performance Computing, Networking, Storage and Analysis, 2014, pp. 217–228.