TomusBlobs: Scalable Data-intensive Processing on Azure Clouds

Alexandru Costan, Radu Tudoran, Gabriel Antoniu, Goetz Brasche

CONCURRENCY AND COMPUTATION: PRACTICE AND EXPERIENCE
Concurrency Computat.: Pract. Exper. 2013; **00**:1–25
Published online in Wiley InterScience (www.interscience.wiley.com). DOI: 10.1002/cpe

TomusBlobs: Scalable Data-intensive Processing on Azure Clouds

Alexandru Costan[1]*, Radu Tudoran[1], Gabriel Antoniu[1] and Goetz Brasche[2]

[1]*Inria Rennes - Bretagne Atlantique, Campus de Beaulieu, 35042 Rennes, France*
[2] *Microsoft Advanced Technology Labs Europe, EMIC, Ritterstrasse 23, 52072 Aachen, Germany*

SUMMARY

The emergence of cloud computing has brought the opportunity to use large-scale compute infrastructures for a broader and broader spectrum of applications and users. As the cloud paradigm gets attractive for the ”elasticity” in resource usage and associated costs (the users only pay for resources actually used), cloud applications still suffer from the high latencies and low performance of cloud storage services. As Big Data analysis on clouds becomes more and more relevant in many application areas, enabling high-throughput massive data processing on cloud data becomes a critical issue, as it impacts the overall application performance. In this paper we address this challenge at the level of cloud storage. We introduce a concurrency-optimized data storage system (called TomusBlobs) which federates the virtual disks associated to the Virtual Machines running the application code on the cloud. We demonstrate the performance benefits of our solution for efficient data-intensive processing by building an optimized prototype MapReduce framework for Microsoft’s Azure cloud platform based on TomusBlobs. Finally, we specifically address the limitations of state-of-the-art MapReduce frameworks for reduce-intensive workloads, by proposing MapIterativeReduce as an extension of the MapReduce model. We validate the above contributions through large-scale experiments with synthetic benchmarks and with real-world applications on the Azure commercial cloud, using resources distributed across multiple data centers: they demonstrate that our solutions bring substantial benefits to data intensive applications compared to approaches relying on state-of-the-art cloud object storage.

KEY WORDS: Big Data; cloud computing; data-intensive processing; cloud storage; MapReduce; scientific applications; Azure

1. INTRODUCTION

Data-intensive computing is now starting to be considered as the basis for a new, forth paradigm for science [17]. Two factors are encouraging this trend. First, very large amounts of data either captured by instruments or synthetically generated by simulation models are becoming available in more and more application areas. Second, the infrastructures allowing to persistently store these data for subsequent sharing and processing are becoming a reality, even if they are still in an emerging phase. These two catalysts (finally!) make it possible to unify pieces of knowledge acquired through the previous three paradigms for scientific research (theory, experiments and simulations) with vast amounts of multidisciplinary data. The technical and scientific issues related to this context have globally been designated as the ”Big Data” challenges and have been identified as highly strategic by the major research agencies, who have recently decided to allocate substantial funding to research projects in this area.

In this landscape, the infrastructure for data storage and processing is obviously the critical piece. Building a functional infrastructure able to properly address the requirements of Big Data

*Correspondence to: Inria Rennes - Bretagne Atlantique, Campus de Beaulieu, 35042 Rennes, France. E-mail: alexandru.costan@inria.fr

applications in terms of data storage and processing remains an important challenge. An important step forward has been made thanks to the emergence of cloud infrastructures, which are bringing the first bricks designed to cope with the challenging scale associated to the Big Data vision. To take an illustrative example, the Amazon cloud is providing the storage support for the data produced by the *1000 Genomes* project [1], which aims to sequence the genomes of a large number of people, to provide a comprehensive resource on human genetic variation. These data have been made available to the worldwide scientific community in a free way on the Amazon cloud infrastructure and can be processed by researchers using the Amazon EC2 computing utility.

Cloud technologies bring to life the illusion of a (more-or-less) infinitely scalable infrastructure managed through a fully outsourced ICT service that allows the users to avoid the overhead of buying and managing complex distributed hardware. Users "rent" those outsourced resources according to their needs from providers who take the responsibility for data availability and persistence. Whereas this model is particularly appealing, it must be acknowledged that cloud technologies have not yet reached their full potential: many capabilities are still far from being exploited to a full degree. In particular, the capabilities available today for storing and accessing cloud data are still rudimentary and far from meeting the application requirements. Many basic issues have been addressed and general data management support is available, however handling massive heterogeneous data is still complicated. Appropriate services for consistent data sharing among multiple cloud users across multiple Virtual Machines (VMs) are still missing and so are the mechanisms for efficiently supporting massively concurrent accesses to data. For instance, there is no such notion of efficient parallel file system on today's reference commercial clouds, which provide object stores such as S3 or Azure Blobs accessed through high-latency REST interfaces both by cloud users and by VMs.

These challenges exhibited by the requirements of data-intensive applications are precisely the ones we address in this paper. More specifically, we typically consider applications such as data analytics, consisting of a very large number of tasks that need to process a large dataset to extract meaningful information. As each individual task usually runs on a separate VM in the cloud, such applications need a high performance storage system that would allow VMs to concurrently access shared data. Using state-of-the-art cloud object stores in the way the application would use a more traditional parallel file system is not feasible for efficiency and scalability reasons: in today's cloud architectures computational nodes are separated from the storage nodes and communication between the two exhibits a prohibitively high latency due the aforementioned data access protocols. Moreover, users need to pay an additional cost for storing and moving data into/out of these repositories. To address this issue, we propose an architecture for concurrency-optimized PaaS-level cloud storage leveraging virtual disks: TomusBlobs. For an application consisting of a large set of VMs, it federates the local disks of those VMs into a globally shared data store. Moreover, this approach requires no changes to the application nor to the cloud middleware. Furthermore, it does not use additional resources from the cloud, as the virtual disk is implicitly available to the user for storage, without additional costs. We implemented this approach in the Microsoft Azure [4] cloud platform. The proposed approach is interesting as the Platform-as-a-Service nature of Azure makes it difficult or rather impossible for the cloud user to set up an existing general purpose runtime on its VM instances: there is no possibility to deploy a parallel filesystem like HDFS [8] or PVFS [11].

Besides efficient storage, data-intensive applications also need appropriate distributed computing frameworks to harness the power of clouds easily and effectively. In this respect, options are rather limited on today's commercial clouds. On Microsoft's Azure cloud (which we consider as a representative commercial cloud for our study), there is little support for parallel programming frameworks: neither MPI, nor Dryad are available, only some incipient efforts to provide Hadoop are in progress. Very recently, a MapReduce runtime was proposed, built on top of the Azure BLOBs for data storage and on the Azure roles model of the VM instances (Web Role/Worker Role) for computations: AzureMapReduce [15]. This framework however involves costly accesses from VMs to BLOB object storage available with Azure, whose efficiency is not satisfactory in a data-intensive context, as explained above. In contrast, we devised an alternative prototype MapReduce

DOI: 10.1002/cpe

framework called TomusMapReduce, which specifically leverages the benefits of our TomusBlobs storage system and efficiently exploits data locality when storing and accessing application data.

Finally, let us stress that *data sharing across multiple applications deployed on multiple cloud data centers* is a major issue that still needs to be addressed. State-of-the-art object storage solutions (such as Azure BLOBs or S3 objects) have not really been designed to cope with access scenarios where data is shared by multiple applications or by multiple services. The adequacy of these solutions in terms of performance is even weaker when those applications or services are geographically distributed across multiple data centers. We also bring a contribution in this direction by integrating this requirement in the design of our bricks (TomusBlobs and TomusMapreduce) in order to enable data sharing and processing among VMs that are geographically distributed through multiple deployments across multiple data centers.

Our contributions can be summarized as follows:

- We introduce a PaaS-level concurrency-optimized cloud storage system for application data that federates the virtual disks in a cloud: TomusBlobs (Section 4).
- We demonstrate the benefits of TomusBlobs for efficient MapReduce data analytics processing by building a prototype MapReduce framework called TomusMapReduce, able to efficiently leverage the properties provided by TomusBlobs (Section 5).
- To specifically address the needs of reduce-intensive applications, we propose an extension of the MapReduce model called MapIterativeReduce together with an architecture for its implementation and we show that this extensions brings substantial benefits to this target application class (Section 5.1).
- We introduce mechanisms enabling MapReduce data analytics to be efficiently performed on data that are geographically distributed across multiple cloud data centers (Section 5.2).
- We evaluate our approach through large-scale experiments with a real application in Azure *across multiple geographically distributed data centers on more than 1000 cores* (Section 7).

Relationship to previous work. This paper extends several contributions introduced in previous papers [28, 26] by putting them into a more global perspective and by showing how each of them contributes to the central goal of filling some major technological gaps in the area of cloud data storage and processing for data-intensive applications, gaps identified as such by the scientific community. Beyond this synthetic work, this paper makes a new step further by addressing the important issue of large-scale data processing on geographically distributed cloud sites: we discuss and demonstrate how the corresponding requirements can be addressed in the design of MapReduce frameworks. Most experimental evaluations presented in this paper have been realized in such a setting and represent new contributions.

2. DATA-INTENSIVE PROCESSING ON CLOUDS: WHERE WE ARE AND WHAT IS MISSING

The cloud concept had started to be advertised since several years and has increasingly captured attention first in industry, because of the high commercial interest associated to it, then in the academia, as many technological and research issues were uncovered. At this point, cloud systems are still at an experimental stage, even if many basic mechanisms are becoming more and more available. This is particularly true for data management mechanisms. It is explained by the fact that, on one side, existing solutions designed in the past for distributed storage have never had to cope with the challenging data volumes and data processing throughput associated to the emerging Big Data context; on the other side, existing cloud storage solutions now in operation have mainly focused on securing basic functionality in a reliable way (e.g. simple object storage without concurrent access to data). If this can be understood from a commercial perspective where stakeholders need to rapidly position themselves in the promising cloud landscape, it clearly appears now that more sophisticated mechanisms are still missing, in order to properly cope with the needs

DOI: 10.1002/cpe

of complex data-intensive applications that are progressively emerging in the Big Data landscape. In this context, we can identify several technological issues that still need to be covered.

Low-latency, high-throughput access to data under heavy concurrency. State-of-the art cloud storage systems are designed with a simple scenario in mind: individual users store non-shared objects, i.e. objects that are not supposed to be accessed by multiple users. In many scientific applications, a significant part of the processing, with a great impact on the overall execution time, is actually spent in accessing the data. Such applications often consist in a set of processes that *concurrently access shared data*. In such scenarios it is important to enhance existing cloud storage systems with means allowing concurrent access to shared data while avoiding the usually high costs of traditional synchronization mechanisms, in order to provide low-latency, high-throughput access to data.

Fine grain access to pieces of Binary Large Objects (BLOBs). Storing small-size records in separate little files is both unfeasible and difficult to manage, even if the storage layer would support it (currently allowed by the API of state-of-the-art cloud storage systems). As an alternative, data sets can be packed together in huge files. In order to exploit the high parallelization of the computation layer, as in the MapReduce model, applications must process small parts of these huge files concurrently. Consequently, an important challenge is to enable the storage layer to provide efficient fine-grain access to files.

Up-and-down scalability. The storage infrastructure needs to be able to efficiently leverage a large number of independent storage resources (e.g., virtual disks) by dynamically aggregating their capabilities in an elastic way, according to the application needs.

Transparency. Managing huge amounts of distributed data (storing, localizing, aggregating, updating etc.) is complex and should not be handled explicitly. The data management system should automatically and transparently handle these aspects offering to applications a uniform view of the whole storage system. It is important to study to which extent traditional techniques designed for smaller scales with the same purposes (e.g. in the field of Distributed Shared Memory systems) could be leveraged, extended and adapted to the context of the unprecedented scale brought to reality by cloud infrastructures.

Efficient support for reduction-intensive workloads. An increasingly important class of data analytics cloud applications require *reduction* operations for aggregation, filtering, numerical integration, Monte Carlo simulations, etc. These algorithms have a common pattern: data are processed *iteratively* and aggregated into a *single final result*. The existing distributed processing runtimes lack explicit support for reduction or implement it in an inadequate way for clouds: they typically do not directly support full reduction into a single file. To achieve this, programmers must implement an additional aggregator that collects the output data from all reduce jobs and combines them into a single result. For workloads with a large number of reducers and large data volumes, this approach can prove inefficient. Some iterative MapReduce frameworks were proposed, but they could hardly be used to tackle this problem efficiently: multiple MapReduce iterations with identity map phases would then be needed. This leads to an extra overhead due to loop control mechanisms and to job orchestration. In contrast, message passing runtime systems such as MPI provide support for reduction through a rich set of communication and synchronization constructs. However, they suffer from little fault tolerance support, which reflects on the applications' perceived reliability on clouds, mainly built on commodity hardware, with failures being rather the norm than the exception. Moreover, there is little support for such parallel programming frameworks on clouds (on Azure in particular).

Large-scale data sharing across multiple data centers. In [19], it is stated that with the increase of job sizes, the classical single cluster MapReduce model becomes *increasingly inadequate*. Hence, the authors propose a set of scheduling algorithms for processing data across multiple clusters using a model derived from MapReduce: Map-Reduce-Global Reduce. Taking into

DOI: 10.1002/cpe

account that data can be acquired from different geographical locations and be further processed on other sites, enabling efficient data storage and processing across multiple data centers becomes a strong requirement for such complex scenarios. Let us stress that object storage services currently available on today's commercial cloud platforms do not provide support for such scenarios.

The work presented in this paper tries to address the above challenges by integrating the aforementioned requirements in the design of the storage system and of the data processing framework. The following sections detail our approach.

3. DESIGN PRINCIPLES FOR SCALABLE CLOUD STORAGE FOR DATA-INTENSIVE APPLICATIONS

To support efficient processing of large amounts of data in clouds, our analysis led us to identify a set of core design principles for the design of scalable cloud storage systems.

Data locality. Accessing data from remote locations increases the cost of processing data into the clouds (both financial and computing time). At the same time, in many cloud deployments the disks locally attached to the VMs (with storage capacities of hundreds of GBs) are not exploited to their full potential. Therefore, we propose to aggregate parts of the storage space from the virtual disks in a shared common pool that is managed in a distributed fashion. This pool is used to store the application level data. For scalability reasons, they are stored in a striped fashion, i.e. split into small chunks that are evenly distributed among the local disks of the storage. Each chunk is replicated on multiple local disks in order to survive failures. With this approach, read and write access performance under concurrency is greatly enhanced, as the global I/O workload is evenly distributed among the local disks. Furthermore, this scheme reduces latencies and has a potential for high scalability, as a growing number of VMs automatically leads to a larger storage system, which is not the case with the default cloud storage service. In turn, it eases the pressure on the latter, which can provide improved performance and quality of service guarantees for the legacy cloud applications that are specifically designed to make use of it.

No modification of the cloud middleware. Our approach targets the commercial / public clouds: it is therefore mandatory that its building blocks do not require any additional privileges. Since the storage system is deployed inside the VMs, the cloud middleware is not altered. Previous works on aggregating the physical disks attached to the compute nodes [22] impose modifications to the cloud infrastructure, so they only worked with open source cloud kits. Our solution is suitable for both public and private clouds, as no additional privileges are required. In general, scientists prefer using clouds as standard users, rather than dealing with the burdens related to the management of a customized cloud deployment.

Loose coupling between storage and applications. Our approach on cloud data management is mainly targeted at (but not limited to) MapReduce applications. We propose a modular, stub-based architecture, easily adaptable to other processing paradigms (e.g. from MapReduce - Section 5 to MapIterativeReduce - Section 5.1 and to Multi-Site processing - Section 5.2).

Genericity. We aim to propose a generic, stand-alone storage solution for data intensive processing in Azure, that is not dependent on a particular distributed file system implementation. With new data processing frameworks appearing at a high rate, several storage solutions can be used to exploit their particularities. A generic data management framework allows to plug in all these storage solutions as black-boxes, on the fly.

No centralized control. When scaling up the computation / storage to a large number of resources, a centralized control of the data flow becomes soon a bottleneck, especially in the cloud where the high-bandwidth is not typically the norm. We choose to address the coordination between

DOI: 10.1002/cpe

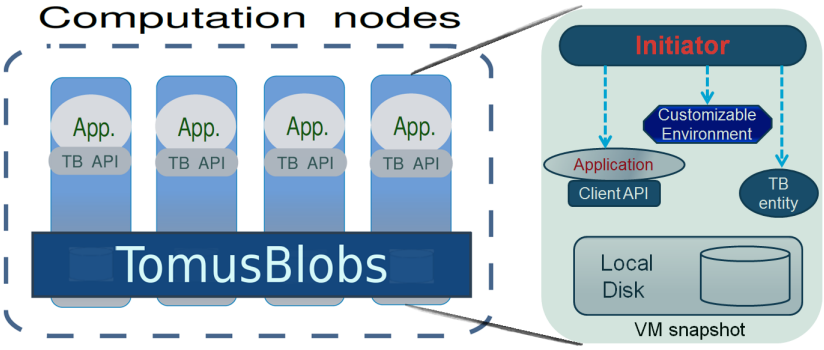

Figure 1. The TomusBlobs architectural overview

the components and the iteration control in a distributed fashion, without any centralized supervision.

4. LEVERAGING VIRTUAL DISKS TO BUILD CONCURRENCY-OPTIMIZED GLOBAL STORAGE: TOMUSBLOBS

Following these design principles, we propose TomusBlobs, a system for *concurrency-optimized PaaS-level cloud storage leveraging virtual disks*. For an application consisting of a large set of VMs, it federates the local disks of those VMs into a globally-shared data store. We rely on the local disk of the VM instance directly in order to share input files and save the output files or intermediate data. This approach requires *no changes to the application nor to the cloud middleware*. Furthermore, it does not use any additional resources from the cloud, as the virtual disk is implicitly available to the user for storage, without any additional costs. We implemented this approach in the Microsoft Azure [4] cloud platform.

The architecture of TomusBlobs consists of three loosely-coupled components presented in Figure 1:

- The *Initiator* component is particular for each cloud. It has the role to deploy, setup and launch the data management system and to customize the scientific environment. It exposes a generic stub that can be easily implemented and customized for any cloud. The Initiator supports the system's elasticity, being able to scale up and down the computing platform at runtime, by integrating the new added nodes in the system or by seamlessly discarding the deleted ones.
- The *Distributed Storage* has the role of aggregating the virtual disks into a uniform shared storage, which is exposed to applications. It is generic as it does not depend on a specific storage solution. Any distributed file system capable to be deployed and executed in a cloud environment (and not changing the cloud middleware) can be used as a Distributed Storage.
- The *Client* consists of the API layer through which the storage is accessed by the applications. Data manipulation is supported transparently through a set of primitives. The interface is similar to the ones of commercial public clouds (Azure BLOBs, Amazon S3).

The local storage of VMs on which we rely consists of virtual block-based storage devices that provide access to the physical storage. The virtual disks appear as devices to the virtual machine and can be formatted and accessed as if they were physical devices. However, they can hold data only for the lifetime of the VM. After the VM is terminated they are cleared. Hence, it is not possible to use them for long-term storage since this would mean leasing the computation nodes for long periods. Instead, we have designed a simple checkpoint mechanism that backups all the data from the TomusBlobs ephemeral storage to the persistent Azure BLOBs, at configurable time intervals (the default being 1h). This backup is done as a background job, privileging the periods with little / no network transfers of the application and remaining non-intrusive (it adds a 4% computational

DOI: 10.1002/cpe

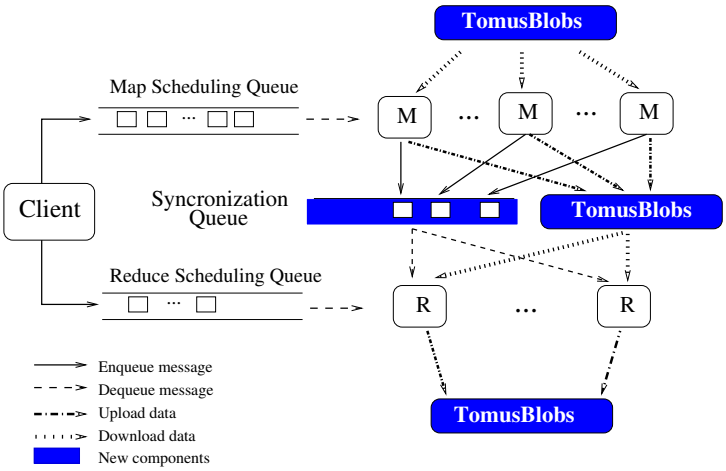

Figure 2. The TomusMapReduce architecture

overhead). Data is restored manually from Azure BLOBs in case of failures at the beginning of new experiments.

Data striping and replication is performed transparently on BLOBs, therefore eliminating the need to explicitly manage chunks in the *Distributed Storage*. The default data management system which we integrated within the Storage uses an efficient version-oriented design that enables lock-free access to data, and thereby favors scalability under heavy concurrency (multiple readers and writers or multiple writers concurrently). Rather than updating the current pages, each write generates a new set of pages corresponding to a new version. Metadata is then generated and "weaved" together with the old metadata in such way as to create the illusion of a new incremental snapshot; this actually shares the unmodified pages of the BLOB with the older versions. Thanks to the decentralized data and metadata management, this provides high throughput and deals transparently with node failures.

5. LEVERAGING TOMUSBLOBS FOR EFFICIENT MAPREDUCE PROCESSING: TOMUSMAPREDUCE

Besides the efficient storage, data-intensive applications also need appropriate distributed computing frameworks to harness the power of clouds easily and effectively. In this respect, options are rather limited on today's commercial clouds. On Microsoft's Azure cloud in particular, there is little support for parallel programming frameworks. The AzureMapReduce framework [15] recently proposed builds a MapReduce runtime on top of the high latency Azure BLOBs for data storage. We devised an alternative prototype MapReduce framework - TomusMapReduce - which specifically leverages the benefits of our storage system. TomusMapReduce relies on TomusBlobs to store input, intermediate and final results, collocating data with computation (mappers and reducers). It uses a simple scheduling mechanism based on the Azure Queues to ensure coordination between its entities. With the storage and computation in the same virtualized environment, high throughout, protection and confidentiality are enhanced, as opposed to the remote Azure BLOBs.

Figure 2 presents the architecture of TomusMapReduce. Clients are provided with a web front-end through which the jobs are submitted. After setting the parameters of the computation, the job descriptions are sent as small messages via 2 queues to the mappers (the *Map Scheduling Queue*) and reducers (the *Reduce Scheduling Queue*). We have also designed a synchronisation mechanism between mappers and reducers using a third queue, the *Synchronization Queue*, used by the mappers to inform the reducers about map jobs' completion. The mappers and the reducers are implemented as a pool of Azure roles that wait for jobs to be assigned.

Building on TomusBlobs, we were able to provide several functionalities needed by scientific applications running on MapReduce platforms. While Hadoop does not support *runtime elasticity*

DOI: 10.1002/cpe

(working with a fixed number of mappers and reducers), our solution seamlessly supports scaling up and down the number of the processing entities. The MapReduce engine is deployed and configured using the same mechanism of TomusBlobs (i.e. by the Initiator). When scaling the deployment, the Initiator is able to update the parameters of the MapReduce framework using the information retrieved by interrogating the cloud middleware (e.g. the number of reducers parameter used for hashing the map's results is dynamically updated and allows elastic scaling).

We address *fault tolerance* both at data level, using the replication support of TomusBlobs, and at task level, using the Azure Queues. We rely on the visibility timeout of the queues to guarantee that a submitted message will not be lost and will be eventually executed by a Worker. In Azure, the messages read from the queues are not deleted, but instead hidden until an explicit delete is received after a successful processing. If no such confirmation arrives, the message will become visible again in the queue, after a predefined timeout. Hence, the job will be either executed and afterwards explicitly deleted or it will become visible again and will be executed by another machine. With this approach, the scheduling process of our framework is protected from unexpected node crashes.

5.1. MapIterativeReduce

The default MapReduce model can be used in numerous scenarios, but when it comes to data processing flows that require a unique result, the model reaches its limitations. Many data intensive applications require *reduction* operations for aggregation, filtering, numerical integration, Monte Carlo simulations, etc. Most of them can be found in domains such as statistics, linear regression, dimension reduction, data filtering, aggregation or mining. These algorithms have a common pattern: data are processed *iteratively* and aggregated into a *single final result*. While in the initial MapReduce proposal the reduce phase was a simple aggregation function, recently an increasing number of applications relying on MapReduce exhibit a reduce-intensive pattern, that is, an important part of the computations are done during the reduce phase. Examples of such applications are to be found in, but not limited to, bio-informatics where a unique result must be computed from a large number of intermediate data (e.g. BLAST, machine learning algorithms, classifiers, etc.). MapIterativeReduce specifically targets such classes of applications.

The existing distributed processing runtime engines lack explicit support for reduction or implement it in an inadequate way for clouds: they typically do not directly support full reduction into a single file. One can either use a single reducer and so the reduce step will loose its parallelism or can create an additional aggregator that will collect all the results from the reducers and combine them into a single result. For reduce-intensive workloads this final operation can become a bottleneck.

We propose MapIterativeReduce, a *new execution model* for reduce-intensive workloads that extends TomusMapReduce and exploits the inherent parallelism of the reduce tasks. In MapIterativeReduce, *no barriers are needed between the map and the reduce phases*: the reducers start the computation as soon as some data is available, reducing the total execution time. Our approach builds on the observation that a large class of scientific applications require the same operator to be applied to combine data from all nodes. In order to exploit any inherent parallelism in reduction this operator has to be at least associative and/or commutative. Most operators that are used in scientific computing (e.g. max, min, sum, select) are both associative and commutative. Reduction may be also used with non-associative and non-commutative operations but offers less potential parallelism.

With the proposed execution model, the typical data flow is presented in Figure 3. This consists of a classical *map* phase followed by an *iterative reduce* phase. The reducers apply the associative operator to a subset of intermediate data, issued either by the mappers or by other reducers from previous iterations. After the computation, the (partial) result is fed back as input to other reducers. These iterations continue until a reducer combines all the intermediate data into a unique result. All reducers check in a distributed fashion whether their output is the final result, using several parameters attached to the reduce tasks and thus avoiding the single point of failure represented by a centralised control entity.

DOI: 10.1002/cpe

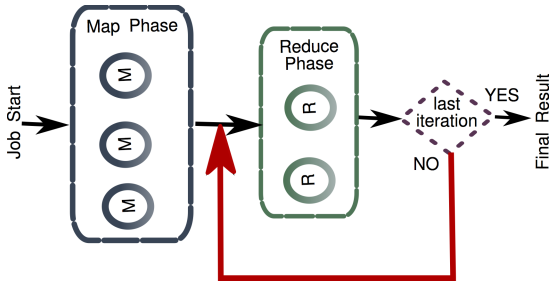

Figure 3. The MapIterativeReduce conceptual model

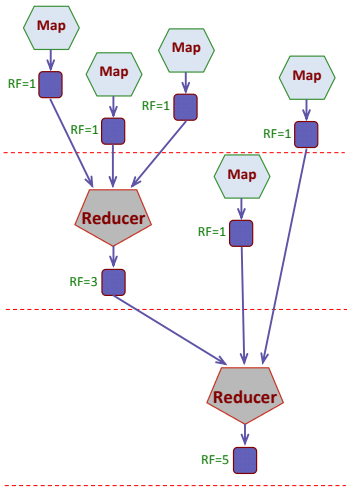

Figure 4. An example of a reduction tree for 5 mappers and a reduction ratio of 3

We can now define the reduction as a scheduling problem: mapping the tasks to the reducers using a *reduction tree*. The iterative reduce phase can be viewed as a reduction tree, each iteration corresponding to a tree level. A clarifying example of an unfolding of the iterative process is given in Figure 4. Each reducer processes 3 sets of data at a time and the initial number of mappers is 5. During the first iteration, only one reducer has all the input available, while in the second iteration the bottom reducer can process the 2 remaining data sets and the data issued from the first iteration to produce the final result. With the associative operator, it is possible to interleave the computation from different iterations to better exploit parallelism.

Two parameters control the scheduling process and the unsupervised termination condition:

The Reduction Ratio - defines the workload executed by a reducer (i.e. the number of intermediate results to be reduced at a time). This parameter and the number of map jobs completely determine the number of reduce jobs that are needed in the iterative process and the depth of unfolding reduction tree. In the previous example, the Reduction Ratio was 3, and in conjunction with the number of map jobs (5) determines the number of iterations (2) and the number of reducers (2). Unlike the collective operations (e.g. in MPI) with binary operators, we are able to support n-reductions.

The Reduce Factor - defines the termination condition, checked locally by each reducer. Since the goal was to process all data into a single result, the termination condition is: *are all the results reduced?*. The Reduce Factor (RF) is used to measure the progress towards this goal. The initial intermediate data (coming from mappers) have a Reduce Factor of 1, the intermediate results produced by a reducer will have a RF equal to the sum of the RFs of all the inputs. The termination condition becomes a comparison between the resulting factor and the total number of jobs. In the previous example, the top reducer has a resulting Reduce Factor of 3 while the last one will have a factor of 5 - equal to the number of mappers.

MapIterativeReduce is made up of several entities, discussed bellow:

1. **The Client** is used to submit jobs (i.e. the description of the workflow). Based on the specified number of mappers and on the selected $ReductionRatio$, it computes the number of reducers and then enqueues all the jobs.
2. **The Mapper** is a classical map task. It will poll for jobs and then download the data, process it and advertise the termination via a queue message.
3. **The Reducer** retrieves a job description submitted by the client, which contains the $ReductionRatio$ and then process that amount of intermediate results as soon as they become

DOI: 10.1002/cpe

Algorithm 1 The Client actions

```
function CLIENT(NrOfMappers, ReductionRatio)
    for i ← 1, NrOfMappers do
        Enqueue_Map_Message(i);
    end for
    reducers ⇐
Create_Reduce_Jobs(ReductionRatio, NrOfMappers)
    for i ← 1, NrOfMappers do
        Enqueue_Reduce_Msg(reducers[i].ReductionRatio,
NrOfMappers);
    end for
    Wait_Final_Results();
end function
```

Algorithm 2 The Mapper operations

```
function MAPPER
    while true do
        msg ⇐ Dequeue_Map_Message();
        if msg then
            Fetch_Data();
            Map();
            Write_Intermediate_Data();
            Enqueue_Sync_Message(1);
        end if
    end while
end function
```

available. Once the processing is over, it tests the termination condition and will either enque partial results so that other reducers in the following iterations will further process it, or it will notifiy the end of the computation.

MapIterativeReduce is implemented as an extension of TomusMapReduce, building essentially on the same set of loosely coupled components with no single point of failure. The Azure Queues are used as a scheduling mechanism and for synchronization between Mappers and Reducers. In the iterative reduce phase we rely on two such queues: the *Reduce Scheduling Queue* and the *Synchronization Queue*. At deployment time a pool of Reducers is constructed based on user specifications and the available resources. With this approach, the Reducers are reused in different iterations to perform several reduction tasks; they are regularly polling the *ReduceSchedulingQueue* in order to retrieve the messages with the job description (e.g., the amount of data to process). When a task is being assigned to them, the Reducers start listening to the *Synchronization Queue*. They wait until at least one message is sent from the Mappers, notifying the end of a map computation and the availability of the intermediate data. It will then dequeue the message and fetch the corresponding intermediate data. At the end of the task, if there are still data to process, the Reducer enqueues a new message to the *Synchronization Queue* to advertise the new intermediate results and transfers the data to the storage system for further processing. Messages are also extended with the control parameters and the characterizations of the reduce tasks, with little overhead as it will be shown in the evaluation section.

One of the key reasons for MapReduce framework's success is its runtime support for fault tolerance. Our approach for dependability is two folded: on one hand, we rely on the implicit fault tolerance support of the underlying platforms (Azure, TomusBlobs), on the other hand, we implemented specific techniques for dealing with failures in MapIterativeReduce. We use the visibility timeout of the Azure Queues to guarantee that a submitted message will not be lost and

DOI: 10.1002/cpe

Algorithm 3 The Reducer operations

```
function REDUCER
    while true do
        msg ⇐ Dequeue_Reduce_Message();
        if msg then
            NrOfMappers ⇐ msg.NrOfMappers;
            reduce_jobs ⇐ msg.ReductionRatio;
            ReduceFactor ⇐ 0;
            while reduce_jobs do
                msg_inter ⇐ Dequeue_Sync_Message();
                if msg_inter then
                    Fetch_Data();
                    Reduce();
                    reduce_jobs−−;
                    ReduceFactor += msg_inter.RedFactor;
                end if
            end while
            if ReduceFactor == NrOfMappers then
                Write_Final_Result();
            else
                Write_Intermediate_Data();
                Enqueue_Sync_Message(ReduceFactor);
            end if
        end if
    end while
end function
```

will be eventually executed by a Worker, as in TomusMapReduce. For the iterative reduce phase, however, this mechanism is not enough. Since a Reducer has to process several messages from different queues, a more complex technique for recovering the jobs in case of failures is needed. We developed a distributed watchdog that monitors the progress of the Reducers. It implements a light checkpointing scheme by saving the state of the jobs to the file system, in parallel with the reduce processing. In case of a failure of some Reducer, the system will rollback the descriptions of the intermediate data and the reduce job, which will be assigned to another Reducer from the pool. This allows MapIterativeReduce to restart a failed task from the previous iteration, instead of starting the reduction from the beginning.

5.2. Data processing across multiple datacenters

As we move to the world of *Big Data*, single-site processing becomes insufficient. Scientific data is typically collected from several sources and even the data acquired from a single source is distributed on multiple sites for availability and reliability reasons. One of the initial obstacles for the uniform data processing across multiple sites was the founding idea of grid computing, based on the assumption that control over how resources are used stays with the site, reflecting *local* software and policy choices. Leaving control to individual sites was a pragmatic choice but also led to a point beyond which grid computing found it hard to scale. Clouds, in contrast, let users control remote resources. In conjunction with a reliable networking environment, we can now use geographically distributed resources: dynamically provisioned distributed domains are built in this way over multiple sites. Several advantages arise from computations running on such multi-site configurations: resilience to failures, distribution across partitions (e.g. moving computation close to data or viceversa), elastic scaling to support usage bursts, etc.

We extended TomusMapReduce to support data processing across multiple sites (e.g. multiple data centers of a given cloud provider, situated on different continents): data is partitioned, scattered

DOI: 10.1002/cpe

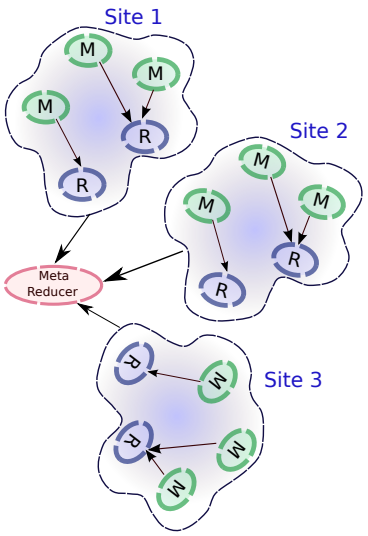

Figure 5. MapReduce across multiple sites

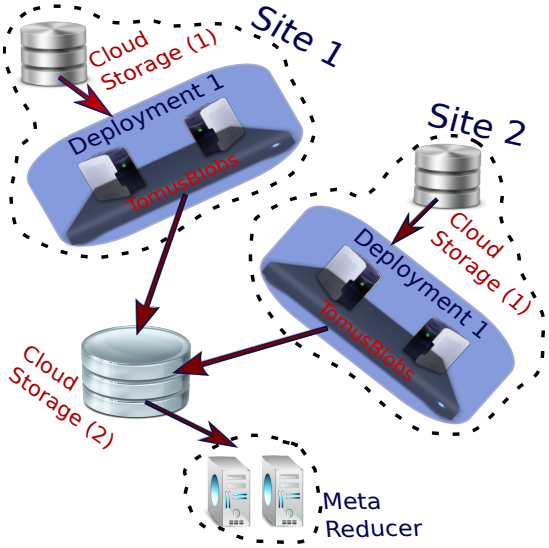

Figure 6. A typical multi-site MapReduce deployment

and processed in parallel, relying on multiple aggregated storage systems to handle it. Several building blocks underpin the creation of a multi-site deployment. In short, we must provide an end-user environment that represents a uniform abstraction - such as a virtual cluster or a virtual site - independent of any particular cloud deployment and that can be instantiated dynamically. We next examine the mechanisms that can accomplish this.

The goal is to setup a virtual cluster large enough to support an application execution and have a tool that is able to handle data across this virtual setup. Rather than simply selecting the site with the largest available resources, we select allocations from a few different domains and build a multi-site environment on top of those allocations. Distinct instances of TomusMapReduce or MapIterativeReduce are deployed at each site, where they run as explained in the previous subsections. To make them work together in a multi-site environment, we enhanced our MapReduce processing framework with an additional entity called *MetaReducer*: its role is to implement a final reduce step to aggregate the results from all sites as soon as they become available. The conceptual architecture is presented in Figure 5.

We use several storage systems to optimize data management within the same deployment or between several deployments, as depicted in Figure 6:

TomusBlobs is used for *intra-deployment* fast data transfers since it supports data locality and provides high-throughput under heavy concurrency. It supports efficient data handling at runtime; data needs to be backed up after the experiment because it relies on *volatile* virtual disks.

Cloud Storage (1) is the remote object storage exposed by the cloud provider on site, typically incurring high latencies and extra costs. It is used *locally* within a deployment for *persistent storage* of the initial and output data.

Cloud Storage (2) is used for *inter-site* data transfers and it relies on the object storage of the cloud site where the MetaReducer is located. Since these transfers are quite expensive in terms of costs and latency, the goal is to minimize them as much as possible. MapIterativeReduce relies on this: it only transfers one final result to the MetaReducer and not the data that still needs to be further aggregated.

We implemented this approach on top of the Azure clouds. We added support for communication with the MetaReducer to the TomusMapReduce and MapIterativeReduce frameworks and designed a scheduling mechanism for scattering the initial data across multiple sites. The MetaReducer was implemented as an independent service, built on top of a pool of reducers. The number of

DOI: 10.1002/cpe

communication queues used is proportional with the number of deployments. Additionally, we used a strategy of hashing data inside the same deployments (when the same data is replicated across multiple storage centers) to minimize data transfers. In the *partition* phase at the beginning of the MapReduce processing, data can be segmented between sites, so that each deployment works on a sub-problem, minimizing the transfers across sites. Data should be replicated across sites or pre-partitioned within the data centers in order to reduce the cost of staging-in the initial data.

6. CASE STUDY: A-BRAIN

We want to assess the impact of TomusBlobs in a real life application: A-Brain is such a joint genetic and neuro-imaging data analysis application involving large cohorts of subjects [2]. This application is used to study and understand the variability that exists between individuals; finding the correlations between brain responses and genetic data may highlight risk factors in target populations. Imaging genetic studies linking functional MRI data and Single Nucleotide Polyphormisms (SNPs) data may face a dire multiple comparisons issue. By genotyping DNA chips several hundred thousands values per subject can be recorded, while in the imaging dimension an fMRI volume may contain 100k-1M voxels (volumetric picture elements). Performing statistically rigorous analyses on such amounts of data represents a challenge as it entails a huge number of hypotheses. Additionally, a correction of the statistical significance of pair-wise relationships is often needed. It is therefore desirable to set up as sensitive techniques as possible to explore where in the brain and where in the genome a significant link can be detected, while correcting for family-wise multiple comparisons (controlling for false positive rate) [24].

Sophisticated techniques need to be used in order to perform sensitive analysis on the targeted datasets. *Univariate* studies find SNPs and neuro-imaging traits that are significantly correlated (e.g. the amount of functional activity in a brain region is related to the presence of a minor allele on a gene). With *regression* studies, some sets of SNPs predict a neuro-imaging / behavioural trait (e.g. a set of SNPs altogether predict a given brain characteristic), while with *multivariate* studies, an ensemble of genetic traits predict a certain combination of neuroimaging traits.

Let (X, Y) be a joint neuro-imaging dataset, acquired in the same population of S subjects. The dataset may also comprise a set Z of behavioural and demographic observations (null for the tests performed in the Section 7). X is assumed to comprise n_v variables, e.g. one for each location in the brain image domain, while Y comprises n_g variables; typically $n_v \sim 10^6$, $n_g \sim 10^6$, when available, Z contains typically less variables (of the order of 10^2). The variables are not independent, as there exist serial correlations between consecutive SNPs and spatial correlations within neighbouring regions. The problem addressed is the detection of statistically significant links between the sets X and Y. This can be performed using the aforementioned methods by testing the statistical significance of the correlation of all (x, y) pairs for $(x, y) \in X \times$ Y. We aim to address these challenges by using TomusBlobs in conjunction with TomusMapReduce / MapIterativeReduce in Azure.

In a first phase, we used univariate analysis to detect the correlations. X counts the number of alleles at a given location in the genome while Y quantifies some brain activity. A possible model (allelic dose model) is:

$$y = \mu + x\beta + z_c\beta_c + \epsilon \tag{1}$$

where μ is a constant, z_c a set of confound variables and β_c the corresponding parameters; β is the parameter to be inferred, which will be finally turned to a significance (p-value). Assuming that the only 3 values for X are (0, 1, 2), a new model can be derived:

$$y = \mu + \beta_0\mathbb{I}_{x=0} + \beta_1\mathbb{I}_{x=1} + \beta_2\mathbb{I}_{x=2} + z_c\beta_c + \epsilon' \tag{2}$$

where the parameters of interest are now the parameters (β_0, β_1, β_2) for the response associated with levels of minor allel frequency. To ensure correction (controlling false detection), the power of the analysis decreases (by a factor $n_v \times n_g$). To address this, a permutation procedure is used, in

DOI: 10.1002/cpe

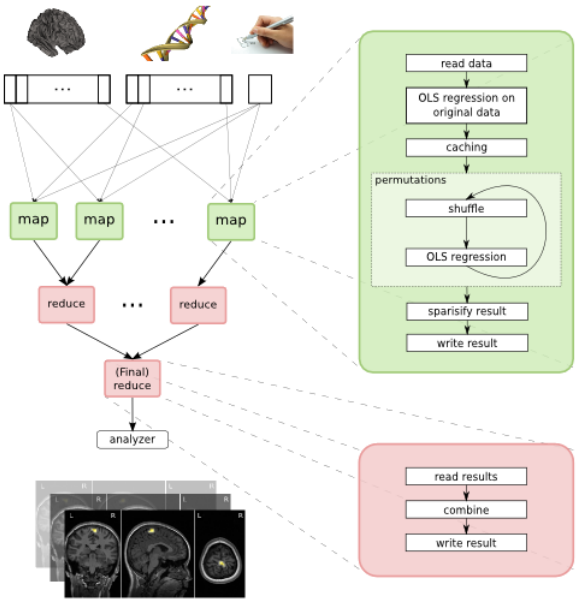

Figure 7. The A-Brain application as a MapReduce process

which the data of one block is reshuffled B times ($\approx 10^4$), and the p-values obtained in the initial regression analysis are compared to those obtained in each of the B shuffles.

A-Brain's challenges in terms of storage and processing power are two folded:

Data - the initial data sets used for testing contain 50K voxels and 500K SNPs for approximately 2000 subjects. However, they are likely to increase, as higher resolution data sets will be available. Following the regression stage, the values of each correlation must be tested and filter the most significant p-values. Taking into account that we use matrices of size $n_v \times n_g$ of doubles (8 bytes), the amount of intermediate data can reach 1.77 PB ($8 \times 10^4 \times 50 \times 10^3 \times 500 \times 10^3$).

Computation - to obtain results with a high degree of confidence, a number of 10^4 permutations is required, resulting in a total of 2.5×10^{14} associations to be computed. The initial univariate algorithm was able to perform 1.67×10^4 association per second, but after significantly tuning the algorithm it now reaches 1.5×10^6 associations per second. On a single core machine the time estimation to run the new algorithm is ≈ 5.3 years. However, the algorithm is embarrassingly parallel, leaving room for improvement if appropriate resources are available.

The application can be parallelized using the MapReduce paradigm (Figure 7). Each mapper takes the same initial data, shuffles it to increase the statistical precision and performs the regression between the 2 sets of data (X, Y) to obtain the correlation between them. This regression represents a series of matrix operations, resulting in a matrix of p-values - the intermediate data. Next, the reduce phase performs a filtering of the intermediate values, generating a unique final result. With this approach, only the correlations with a p-value higher than a specified threshold are considered relevant and kept.

7. PERFORMANCE EVALUATION

In this section we assess the impact of TomusBlobs and the MapReduce frameworks in synthetic settings and for real-life applications. The experiments were performed on the Azure cloud on several hundred nodes in geographically distributed sites, scaling up to *1000 cores*, a premiere for scientific applications running on Azure.

DOI: 10.1002/cpe

7.1. Experimental setup

For our experiments we used 3 types of setups, denoted from now on as *small*, *medium* and *large*. For the small setup we have used up to 100 VMs (small instances), while for the medium setups we used 200 VMs (small instances); in both cases all instances belong to a single deployment. For the large setup we used a multi site deployment scattered across West Europe, North Europe and North US. Each deployment comprised 330 cores, reaching a total of 1000 cores for the global experiment. The small instances of the VMs in Azure have 1 CPU core, 1.75 GB of memory and 225 GB of disk space.

All virtual machines have their own physical CPU. The cloud infrastructure is composed of physical nodes that have 8 cores, hence the same physical machine will be shared among different applications. The strategy used by the cloud fabric controller to place the VMs considers the availability in case of failure, so the VMs will be spread on different racks and different physical machines. The available network connection between the physical nodes is Gigabit Ethernet. However for all VMs the network of the physical machine is shared among applications, while the backbone traffic is shared in all cases.

7.2. Cloud storage evaluation: locally distributed file systems vs. remote object storage

In our first series of experiments we were interested to assess the optimal size of the chunks used to split data by the underlying storage system. We measured the throughput obtained when writing 128 MB of data and increasing the chunk size with TomusBlobs, AzureBlobs - the default object storage of Azure, and its optimized counterpart - using the ”affinity groups”. This affinity concept allows an increased proximity for the co-location of storage and hosted services within the same datacenter. Since this option can reduce latencies for the Azure Storage, we used it when comparing to TomusBlobs. As seen from Figure 8 the highest throughput is attained with a chunk of 4 MB, similar to the default chunk size for the AzureBlobs, so we suspect that the underlying infrastructure is optimized for network packages of this size.

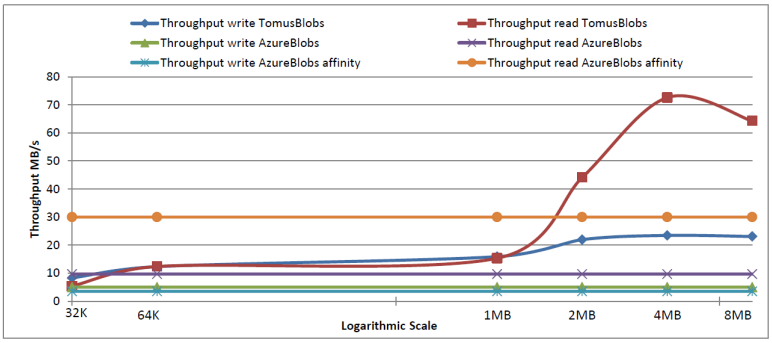

Figure 8. Storage performance with respect to the page (chunk) size

After setting the chunk size, we performed a series of experiments to evaluate the throughput performance of our proposal in controlled synthetic settings. We have implemented a set of microbenchmarks that write/read data in Azure and have measured the achieved throughput as more and more concurrent clients access the storage system. This synthetic setup has enabled us to control several access patterns exhibited by highly-parallel applications. We can thus directly compare the respective behavior of TomusBlobs and Azure Storage in these particular synthetic scenarios using a *small* setup:

- a single client reading and writing a large data object
- concurrent clients reading different parts of the same large data object and writing data to several large objects

Scenario 1: single reader / writer, single data. We first measure the throughput achieved when a single client performs a set of operations on a data set whose size is gradually increasing (limited by

DOI: 10.1002/cpe

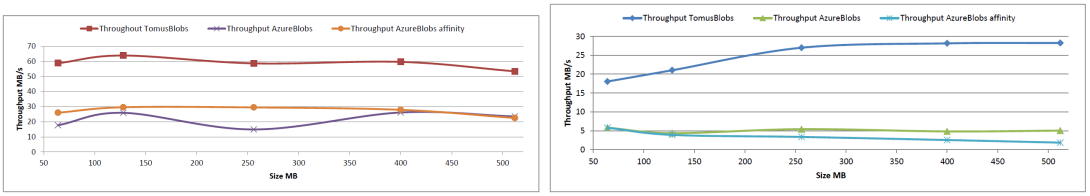

Figure 9. Storage read / write throughput with respect to data size for a client

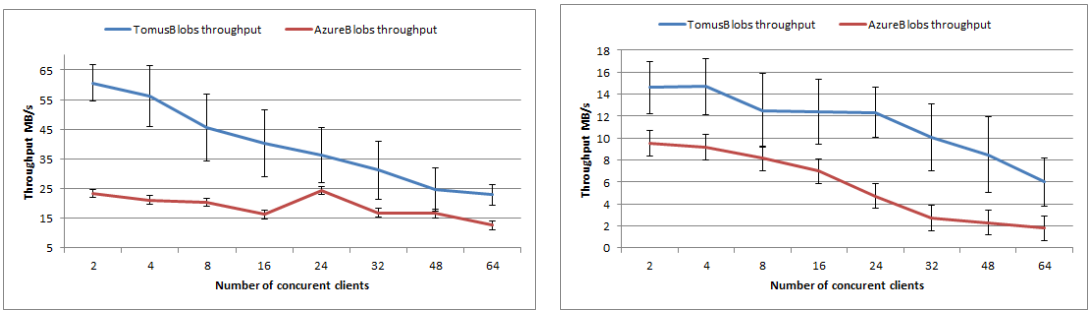

Figure 10. Read / Write throughput per client under concurrent access

the available memory for a small Azure machine). This scenario is relevant for the applications where most of the time each process manipulates its own data set independently of the other processes (e.g. simulations, where the domain is typically distributed among processes that analyze it and save from time to time the local results).

From Figure 9 we notice that TomusBlobs achieves a significantly higher throughput than Azure Blobs (approximately 2 times higher), as a result of using the low latency local disks. Also, the strong consistency model used by Azure Storage (as opposed to the eventual consistency in TomusBlobs) and the fact that they combine the workloads of many different users together to reduce storage usage (which is not the case of TomusBlobs) penalize AzureBlobs' performance. Even with the affinity option on, AzureBlobs cannot compensate for the higher latency of the remote storage.

Scenario 2: multiple readers / writers, single data. In the second series of experiments we gradually increase the number of clients that perform the same operation concurrently and measure the throughput. For each given number of clients, varying from 1 to 65, we execute the experiment in two steps. First, all clients write concurrently 128 MB of random data from memory to the storage. Second, they read it back to the memory. This scenario is relevant for applications where multiple clients concurrently read the input data or write the temporary or final results. In particular, the pattern where multiple readers request data in fine-grain chunks is common in the "map" phase of a MapReduce application, in which mappers read the input in order to parse the (key, value) pairs. Figure 10 shows that TomusBlobs outperforms Azure Blobs almost 3 times for writing while the read throughput is almost 2 times higher. These results motivate the adoption of a locally distributed file system for storing data during the experiments as it allows faster and cheaper data processing.

7.3. Evaluating the classical MapReduce offerings for the Azure cloud

We evaluated the impact of using TomusMapReduce instead of AzureMapReduce, the de-facto MapReduce framework for Azure, which relies on the Azure Blobs for storage, using a single site *medium* deployment and multi site *large* deployments. A first series of experiments focuses on *scalability* by measuring the the total execution time as the number of Mappers progressively increases while the size of the input data remains constant - 1GB (the overall amount of data is approximately 40GB). We scaled the experiments such that all the mappers work in parallel (no

DOI: 10.1002/cpe

mappers are left idle), in order to test the ability of the storage backend to support concurrent accesses. Conceptually this means that more shuffles are performed (each mapper performs a shuffle) in parallel, thus increasing the precision of the univariate analysis. Unlike a typical MapReduce, by increasing the number of map jobs, we do not increase the degree of parallelism for the map phase but rather increase the workload. The data is not split among mappers, instead each map job performs an independent computation on the same data. Each mapper generates its own intermediate results, having a constant size given the initial data. Increasing the number of mappers is equivalent to generating more intermediate results that must be processed by the Reducer.

In Figure 11 we present the total execution time (including the time to retrieve initial data and storing the final results from / to Azure Blobs) of A-Brain with the two frameworks, considering in both cases. Data proximity significantly reduces the completion time with TomusMapReduce, especially for a larger number of mappers, which can read and write in parallel from the virtual disks. We notice that while the map time for AzureMapReduce increases due to the increasing number of jobs executed in parallel, the map time in TomusMapReduce scales well and remains almost constant.

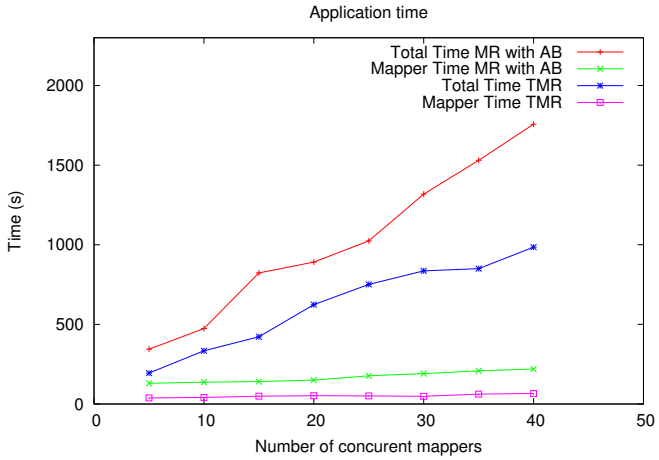

Figure 11. A-Brain execution time when the number of map jobs is varied

Before scaling up the experiments to multi-site deployments we conducted a *medium* size experiment on a single site to get a better understanding of the storage throughput impact on MapReduce frameworks. We have tested the read and write patterns of reducers and mappers (each of them processing 128 MB of data) against different storage solutions: TomusBlobs, AzureBlobs and AzureBlobs with a configuration for multi threading handling of reads/writes, with the number of threads set to 8. The results are presented in Figure 12 and show how data locality impacts the throughput; even when we use CPU cycles for faster data transfers, the cost of accessing a remote public storage is higher than with our approach.

We now change the scale of the experiments to multi-site deployments involving up to 1000 cores. In Figure 14 we analyze the variation of the average map time when the number of jobs is increased while running A-Brain in a *large* setup; the initial data was split and scattered across sites. Although we increase the number of parallel map jobs, the average time remains approximately the same (each mapper processes 100 MB of data, as in all consequent experiments). This shows that the scalability of TomusMapReduce is preserved even when distributing the computation across several sites. We note that the jobs executed in West Europe have slightly larger makespans than the others: this is normal behavior since significant correlation links found with these experiments were mostly located in the first partition of the data (assigned to this data center), generating a higher workload per task.

We now examine the reduce time with the same experimental setup - Figure 13. We vary the number of intermediate results to be reduced and notice that the computation structure in the

DOI: 10.1002/cpe

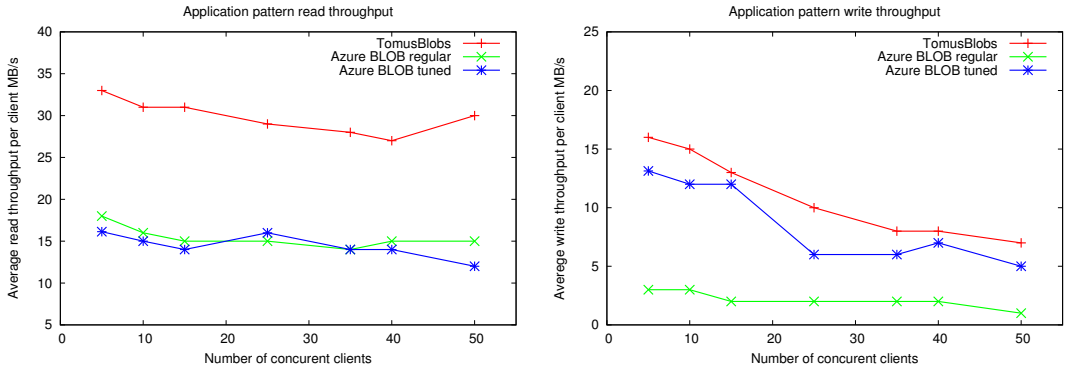

Figure 12. Storage read / write throughput in the context of executing a MapReduce application

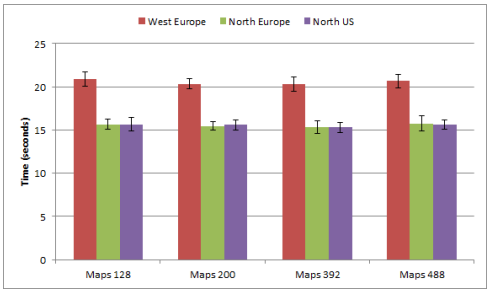

Figure 14. Variation of the average map time on different sites

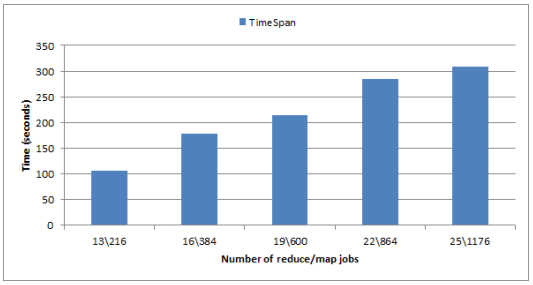

Figure 15. A-Brain application execution time on a multi site deployment of MapReduce engines

reduce phase remains almost the same, a proof of a good scalability with increasing workloads. This translates into predictable performance for the storage access time, with respect to the size of intermediate data, a feature often speculated in high performance computing [13].

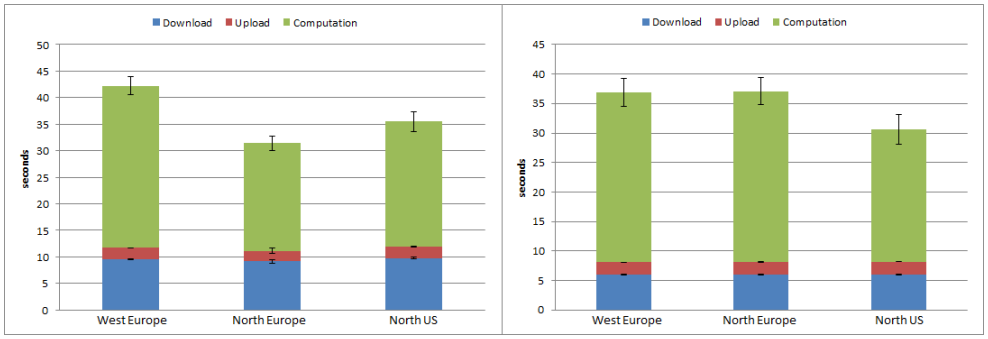

Figure 13. Structure of the reducers processing phases (left - reduction ratio 5 ; right - reduction ratio 3)

We continued by successively increasing the workload and at the same time the number of jobs (both map and reduce jobs) on the *large* setup with 1000 cores. In Figure 15 we notice a good behavior of TomusMapReduce at large scale: A-Brain's execution time only increases 3 times while the workload is increased 5 times.

We now examine how TomusBlobs, the intra-deployment storage, can cope with large workloads. We measure the map / reduce throughput as the workload is increased, putting additional pressure on the storage: the data size is increased from 3 GB up to 15 GB. From Figure 16 we note that TomusBlobs is able to sustain an almost constant throughput due to data and metadata decentralization and the lock-free synchronization with versions.

DOI: 10.1002/cpe

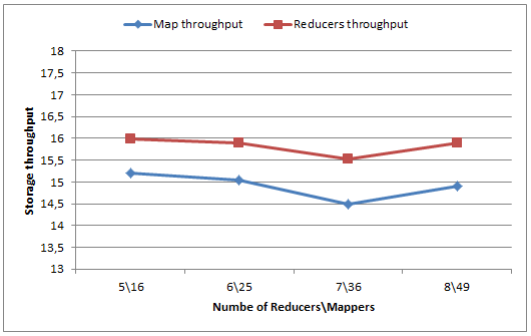

Figure 16. Map / Reduce throughput with an increasing workload

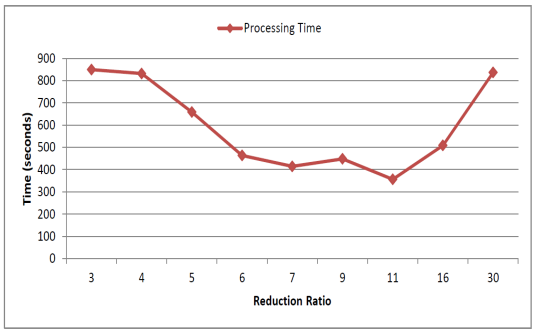

Figure 17. Evaluating the impact of the $ReductionRatio$ on the overall execution time

7.4. Performance gains with MapIterativeReduce

The parameter that has the highest impact on performance is the $ReductionRatio$. The amount of intermediate data that a reducer should process for an optimum performance depends on many factors. One of them is the number of map jobs, as this determines the number of leafs in the reduction tree. However, even for an equal number of mappers, two different applications (i.e., with different computation patterns and data sizes) don't necessarily have the same optimal $ReductionRatio$. Another influencing factor is the tradeoff between the time spent by a reducer for processing data and the time needed to transfer it. To assess its impact on performance we used a *small* setup to reduce (with the minimum operator) the same amount of matrix data, 7.5 GB in total, issued from 30 mappers, while varying the $ReductionRatio$. Measuring the total execution time, in Figure 17, we notice that it decreases with the increase of the $ReductionRatio$ up to a point where the workload assigned for a reducer becomes much higher then the cost of transferring data and the performance drops. An optimal value for the $ReductionRatio$ depends on the application and the size of data. One needs to build a specific model for the application to determine the optimal balance between local computations and data transfers.

We evaluated MapIterativeReduce with a classical MapReduce benchmark: MostFrequentWords - derived from the WordCount example by extending the reduction with a merge sort that combines and orders the words' occurrences. We used a *medium* setup and a $ReductionRatio$ of 5. The amount of data to be processed was successively increased from 3.2 GB up to 32 GB. We compared MapIterativeReduce with TomusMapReduce and AzureMapReduce augmented with a final aggregator that reduces the final results to a unique one in the standard fashion. The results show that when the workload increases becoming more reduce-intensive, our framework outperforms the others, decreasing the computation time up to 40% and 60%, respectively. For small workloads, MapIterativeReduce and the classical MapReduce have similar performance. This is due to the fact that the depth of the reduction tree is small and the latency of the initial transfers between mappers and reducers is predominant in the total execution time. We repeated this experiment with the A-Brain application since it requieres a unique final result: we increased the the number of map jobs from 50 to 500, thus increasing the amount of intermediate data to be reduced from 5 GB to 50 GB. The computation time in this case decreased almost 5 times, as seen in Figure 18.

7.5. Impact of multi-tenancy on performance

Public clouds are subject to multi-tenancy. Cloud providers work on several mechanisms to virtually isolate deployments with the goal of offering fairness to users. In this context, we refer to fairness as getting the performance that you are paying for. However, the question of how multi-tenancy affects performance remains an open issue, which we explore in this subsection.

As all other commercial clouds, Azure offers different types of VMs. The biggest one, *ExtraLarge*, maps directly to the physical machines. For the other types, there is a one to one relationship between virtual cores and physical cores: the physical machine collocates other VMs belonging to different users. There are no interferences at CPU level, but a degradation of

DOI: 10.1002/cpe

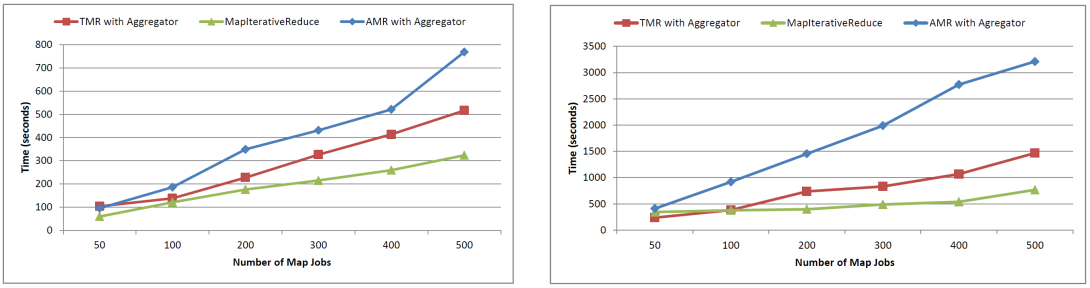

Figure 18. The execution times for Most Frequent Words (left) and A-Brain (right) when scaling up the workload

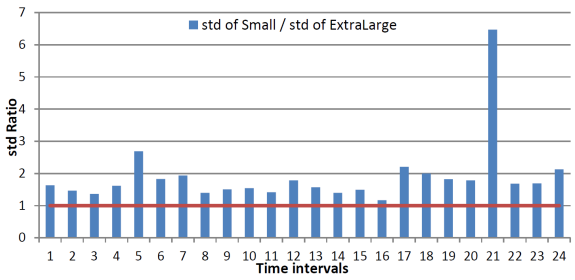

Figure 19. The variation of standard deviation in the context of multi-tenancy[27]

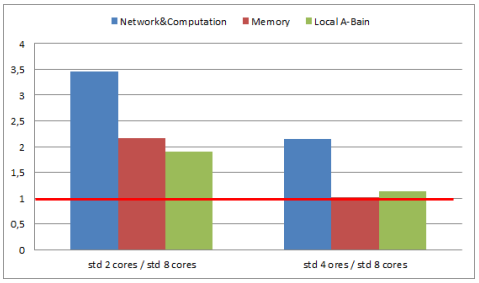

Figure 20. The impact of multi tenancy on different applications (Network & Computation Intensive; Memory Intensive Computation and Local A-Brain)

performance is expected when accessing the local (shared) storage, memory or network. We propose the following methodology: as there is a VM type guaranteed to acquire a physical machine, we will use it as a reference (no multi-tenancy, everything is isolated except the network). We will compare the standard deviation of an application executing in this environment with the same application executed on VMs that might share the physical machines with applications belonging to other users. We assume that there exists a ”default” variability that is due to the OS and to the virtualization mechanisms. Since the ExtraLarge VM occupies the entire physical machine and all computations are done locally, we consider that the standard deviation measured for the ExtraLarge VMs approximates this default variability.

In a first experiment we considered an application derived from A-Brain which executes locally (i.e. reads data from the local disk, performs matrix computation and stores the result back to the local disk). We executed this experiment several days for 1440 times on both ExtraLarge and Large machines, grouped the results in pairs of 60 and computed the standard deviation. The goal of this analysis is to see the fairness of sharing the physical nodes among VMs belonging to different users. A value of the ratio close to 1 (in red) in Figure 19 would mean that either the neighboring applications do not affect our performances, or that there are no neighboring applications, as we can see for the 16th time interval. A high value for the ratio would indicate that the performance of the instance is affected by external factors, as is the case for the 5th and 21th interval. The reason for observing both small and big values is that the samples (2880) were done in a time span of approximately 1 week, one minute apart, a time span that proved sufficiently large to capture various situations. The interferences that affect the instances are caused by the local disk I/O or by memory accesses (we dont include the CPU since there is a 1 to 1 relation between virtual CPUs and physical cores). Such an analysis could be used as a hint for schedulers when distributing the workload among the instances.

In the next experiment we consider 3 different applications to be executed on different VM types. The first one is an application that performs computations and many network transfers (), the second one performs just in-memory operations while the third one is the modified A-Brain, running locally. We have executed these applications on Medium, Large and ExtraLarge VMs. In Figure 20 we

DOI: 10.1002/cpe

represent the ratio between the standard deviation observed on Medium and Large VMs against the one of the ExtraLarge VMs. The highest variation occurs when the (shared) network is accessed. This can be explained by assuming that the network card is subject to bottlenecks more often than the physical memory or the local physical disk. On the other hand, the lowest standard deviation is observed for the in memory operations. So, multi-tenancy mainly affects applications using the network and the local disk. For the latter, the usage of multiple hard disks could be a solution.

7.6. Cost analysis

We are interested to assess the cost of running a representative application like A-Brain in a commercial cloud. We start by dividing the overall execution into 2 parts: computation and transferring data. For expressing the two costs analytically, we assume that O is the overall computation workload to be done, f_n is the fraction of the data analyzed per second if we use N computing nodes and D is the total amount of data manipulated. For the particular case of the univariate analysis of the neuro-imaging data, O represents the number of associations (significant links) to be tested between genes and brain locations, while f_n represents the number of associations per second done by the N machines. Hence the two components of the cost are:

$$cost_{computation} = \frac{A}{f_n} * N * c_{VM},\ where\ c_{VM}\ -\ cost\ of\ a\ compute\ hour$$

$$cost_{data} = D * (c_s + c_t),\ with\ c_s\ -\ cost\ for\ storing\ and\ c_t\ -\ cost\ for\ transfer$$

The cost for storing the data is more complicated than this. Usually it is expressed as a monthly rate, but it is computed based on averages of the data stored during the hours in a day. In applications in which most of the data are transient, like A-Brain, data will be stored only until it is processed and discarded afterwards. We will therefore consider the data cost computed as the overall amount of data multiplied by the cost to store data per month normalized by the simulation time.

$$cost_{data} = D * cost_{hour} \frac{Nr\_of\_hours}{31 * 24}$$

Based on these formulas and on the previous experiments on Azure we compute the storage cost using 200 cores. The average computation speed of the algorithm at this scale is $2.94 * 10^8$ associations per second and the amount of significant data reaches ($\approx 10TB$). The price of one hour of computation is 0.08 euros and the cost for storing 1GB for 1 month is 0.11 euros [6]. We obtain:

$$cost_{total} = \frac{2.5 * 10^{14}}{2.94 * 10^8} * 200 * 0.08 + 10 * 1024 * 0.11 * \frac{\frac{2.5*10^{14}}{2.94*10^8}}{31 * 24} = 4133\ euros$$

This gives an estimate of the cost to process scientific data in commercial clouds for an application representative for a wide class of domains: climate modeling, image processing, protein simulations etc. However, using the storage and data processing platforms that we proposed, this cost can be reduced: assuming an average speedup of 25% with TomusMapReduce and storing data locally with TomusBlobs instead of AzureBlobs, costs are reduced with more than 30% to 2832 euros. The price comparison considers the computation itself without enabling the periodic backup mechanisms of TomusBlobs.

8. RELATED WORK

A number of groups in the research community are investigating how the cloud model might impact the services offered to the scientists and the evolution of the software infrastructures to manage these resources. However, its benefit for serving the needs of scientific applications is still relatively unexplored. While clouds are used today rather as resource pools, in the near future they will be more oriented to hosting services. In this context, there are a number of issues related to scientific applications that are currently explored: clouds typically provide resources, but the software is at

DOI: 10.1002/cpe

the user's responsability, running on multiple nodes may require cluster services (e.g. schedulers), dynamic configuration which is non trivial, efficient support for moving and handling big data, applications need to communicate through files - file systems are needed, appropriate programming paradigms (e.g. MapReduce) which need to be optimized and fine tuned to the characteristics of the underlying clouds.

Many previous works have addressed the performance of scientific applications on virtualized and cloud platforms [23, 10]. Our work is different from these in several ways. First, the existing research mainly focuses on the computing performance of the applications deployed in the cloud. We are rather interested in their data management issues. Second, a broad class of these efforts targets tightly-coupled applications such as MPI-based ones. In contrast, we have focused on loosely-coupled parallel applications with very different requirements (i.e. MapReduce). Third, previous research works have focused mainly on micro benchmarks and benchmark suites. Our work, on the other hand, studies the performance of real-life applications. Nevertheless, there is little progress on optimizing the data management in Azure: to our knowledge there is no distributed file system that can be deployed in the VM nodes; our contribution aims to fill this gap.

The idea of leveraging the locally-available storage on compute nodes to efficiently store and share application-level data is not new. This is one of the founding principles for some scalable distributed file systems, such as Gfarm [22]. This approach provides the abstraction of a global parallel file system which physically integrates the local disks of the compute nodes. For a given access request to data, it favors access to the local disk first if possible; if not, the operation translates into access to remote storage through the network (possibly to local disks of other compute nodes). More recently, in Azure, users were given the possibility to mount the remote storage directly as a virtual drive (e.g. Azure Drives [5]), which allows an easier access to the blob storage.

Such architectures usually assume high-performance communication between computation nodes and storage nodes. This assumption does not hold in the current cloud architectures, which exhibit much larger latencies between compute resources and storage resources. Although a large number of network storage systems exist, few of them can be deployed on clouds. Most of them need special configuration or handling to get them to work in a virtualized environment, while others cannot be executed at all, since they require kernel modifications (e.g. Lustre [25]), which are not allowed by most cloud providers. Depending on the target applications and on the way they interact with data, additional constraints need to be taken into account: typically, object storage systems are not POSIX-compliant [7, 18]. They rather rely on REST type interfaces; if one needs to mount such storage systems and to support standard semantics, additional wrappers are required to make the necessary translations. To the best of our knowledge there is no distributed file system yet that can be deployed by standard users on VMs running on Azure cloud; our contribution aims to fill this gap.

Experiments with several storage and file systems deployed on the *Amazon EC2* clouds are compared in [3] to assess their suitability for executing task workflows. GlusterFS performs well both with a large number of small files and with a large number of clients. NFS proved as a good option only for applications with few clients or low I/O requirements. Both PVFS and S3 scale poorly when the number of small files increases. The results are not surprising in the case of PVFS since the authors used an older version, which does not contain the optimizations for small files included in recent releases. As expected, the choice of storage system impacts significantly the workflow runtime. In comparison, our work is different in scope: 1) we target a PaaS instead of IaaS clouds; 2) we target a different programming model (MapReduce rather than workflows); 3) we complement the performance metrics used by the authors (application's makespan and cost) with I/O related ones (e.g., throughput).

Similarly to our work, the authors of [22] deployed Gfarm in a compute cloud, *Eucalyptus*, using the local physical disks on the compute nodes for storing data. Evaluations with micro benchmarks and with MapReduce applications show scalable I/O performance. With this approach the file system runs on the host OS: this requires to specifically modify and extend the cloud middleware in order to make it aware of the new deployed file system. Whereas this solution can be used with open-source IaaS clouds and on private infrastructures, our approach is different: the storage system

DOI: 10.1002/cpe

can be deployed by the PaaS user within the VMs *without modifying the Cloud middleware* and can thus be used on any cloud infrastructures, including commercial ones.

In [20] the cost of "porting" MapReduce applications to the Cloud is evaluated in order to find a proper trade-off between cost and performance for this class of applications. Several storage backends for the Hadoop framework have been compared: HDFS [8] and BSFS [21] (deployed within the same VMs used for MapReduce) as well as S3. HDFS and BSFS show good performance thanks to data locality, while the S3-based backend, residing on remote storage nodes, forces the Hadoop tasks to read input chunks and to upload their output remotely, which leads to much slower executions. In comparison to this solution where data storage is integrated with Hadoop, we provide here a more general storage solution decoupled from the MapReduce framework. We also show how it is used by a MapReduce prototype on Azure clouds, where Hadoop is not available.

AzureMapReduce [15] is one of the first attempts to address the need for a distributed programming framework in *Azure*. It is a MapReduce runtime designed on top of the Microsoft Azure PaaS cloud abstractions. The architecture leverages the highly scalable Azure services to provide an efficient, on demand alternative to traditional MapReduce clusters. It uses Azure Queues for map and reduce task scheduling and relies on Azure Worker Roles to perform the computations. Our work is similar in this respect. AzureMapreduce, however, does not exploit locality: it uses Azure Tables for metadata and Azure BLOBs for data storage, which results in high access latencies and expensive data transfer costs between Azure storage and VM instances. In contrast, we leverage the VM local disks to exploit data locality and opt for a simpler, efficient communication scheme for the coordination between entities. Also, obtaining a final aggregated result in AzureMapReduce is only possible through an additional aggregator, implemented by the user, that collects and combines the output of all reduce tasks, which may lead to a substantial overhead.

Providing good performance for processing data analytic workflows is an important issue for these MapReduce based platforms, as the system performance could directly map to the monetary cost in the cloud environments. However, the existing MapReduce based data processing frameworks fall short of efficient performance optimization strategies. Until now, optimizations were usually performed manually by users which is a difficult and inefficient task. To address this problem, there are great amount of works focusing on the performance optimization for processing workflows on MapReduce based frameworks in the cloud. The model and its implementations have been extended to better fit a variety of application requirements. Examples of such extensions include interactive analytics, job pipelining and the support for *iterative computations*. In this paper, we focused on this latter direction. Previously proposed iterative frameworks try to cope with reduce-intensive workloads in two ways: 1) by using some additional centralized components to control the convergence of the iterative process; 2) by introducing new programming models. In contrast, our approach maintains the clean programming model of MapReduce and does not add any component for centralized control. Moreover, most of these techniques are orthogonal to our research: as they do not provide explicit support for reduce-intensive workloads, implementing such support incurs an important overhead. Instead, we provided a foundation dedicated to reduce-intensive applications. To our knowledge, MapIterativeReduce is the first work that specifically targets this type of applications on clouds.

A class of research efforts builds *iterative frameworks* targeting applications that reuse a working set of data across multiple parallel operations; a particular use of such iterative processing could be a reduce computation. Spark [29] was recently developed to optimize iterative and interactive computation. It uses caching techniques to improve performance for repeated operations. The main abstraction in Spark is a resilient distributed dataset (RDD), which is maintained in memory across iterations and fault-tolerant. Spark exposes a functional programming interface and can be used interactively as a general-purpose programming language to process large datasets on a cluster. Rather than using a high-level programming model, CloudClustering [12], a framework for iterative data analysis algorithms for Azure clouds, is built directly on the basic services provided by Windows Azure. Its centralized control component - the master - makes the system vulnerable to failures. The architecture of MapIterativeReduce is different of these works that introduce a new programming model to implement a data-flow computation model. Our work relies on the simple

DOI: 10.1002/cpe

MapReduce programming model, extended to optimize the reduce phase, and can be easily adopted by the existing applications without any modification.

Some recent efforts introduce support for *iterative applications* into the MapReduce engine. As Hadoop does not support iterative processing by design, HaLoop [9] was built on top of it with this purpose. It exposes a new programming model and relies on a loop-aware task scheduler and on loop-invariant data caching. Besides HaLoop, other solutions accelerate iterative algorithms by maintaining iteration state in memory. Twister [14] employs a light-weight MapReduce runtime system and uses publish / subscribe messaging-based communication instead of a distributed file system. Mappers and reducers are long-running with distributed memory caches in order to avoid repeated data loading from disks. However, Twister's streaming architecture between mappers and reducers is sensitive to failures, while the long-running mappers and reducers with caches might not scale in commodity machine clusters, where each node has limited memory and resources. The Azure declination of Twister, Twister4Azure [16], the successor of AzureMapReduce, allows optimized intermediate data transfers to cope with these issues: it supports direct fault-tolerant streaming transfers and uses data caching; it adds a merge step to perform final aggregations and shares with TomusBlobs the idea of using the virtual disks as a local cache during application's execution. iMapReduce [30] extracts common features of iterative algorithms and provides support for them. In particular, it relies on persistent tasks and persistent socket connections between tasks, it eliminates shuffling of static data among tasks, and supports the asynchronous execution of iterations when possible. All of these frameworks target applications with iterations *across MapReduce jobs* and require additional components and programming efforts to automatically aggregate their output. MapIterativeReduce specifically addresses this goal by scheduling reduction jobs as soon as their input data is available; map and reduce jobs can thus be interleaved and the usual barrier between these two phases be eliminated.

9. CONCLUSIONS

Porting data intensive applications to the clouds brings forward many challenges in exploiting the benefits of current and upcoming cloud infrastructures. Efficient storage and scalable parallel programming paradigms are some critical examples. This paper has presented TomusBlobs, a cloud storage solution aggregating the virtual disks on the compute nodes, TomusMapReduce, a prototype MapReduce framework relying on it, and MapIterativeReduce - its iterative reduction counterpart, specifically designed to address these challenges. We demonstrated the benefits of our approach through multi-sites experiments on a thousand cores - the largest scientific experimental setup on Azure up to date - using synthetic benchmarks as well as a real-life application. The evaluation shows that it is clearly possible to sustain a high data throughput in the Azure cloud thanks to our low-latency storage: TomusBlobs achieves an increase in throughput under heavy concurrency of up to 5x for writes and up to 3x for reads, compared to Azure Blobs. By using it as a storage backend for our MapReduce framework in A-Brain, we achieved a time speedup of up to 2x compared to state-of-the-art, as well as an increased throughput. MapIterativeReduce builds on the low-latency of TomusBlobs and exploits the inherent parallelism of the reduce tasks. The reduce jobs run iteratively over subsets of the intermediate data, combining them into a single result. In particular, we eliminate the barrier between map and reduce phases, while preserving the cleanness of the MapReduce programming model. We evaluated this framework with a typical MapReduce benchmark achieving speedups ranging from 2 to 4.

In the future, we will design a performance model for TomusBlobs, which considers the clouds variability and provides some optimized deployment configurations. We are also exploring ways of integrating TomusBlobs with other data intensive processing platforms built on Azure, as the Venus-C framework.

ACKNOWLEDGEMENT

DOI: 10.1002/cpe

This work was supported in part by the Agence Nationale de la Recherche under Contract ANR-10-SEGI-001 and by the joint INRIA - Microsoft Research Center. The experiments were carried out using the Azure Cloud infrastructure provided by Microsoft in the framework of the A-Brain MSR-INRIA project. The authors would like to thank the Azure support teams from EMIC for their valuable input and feedback.

REFERENCES

1. The 1000 genomes project. http://aws.amazon.com/fr/1000genomes/.
2. A-Brain, http://www.msr-inria.inria.fr/Projects/a-brain.
3. Amazon web service. http://aws.amazon.com/.
4. Azure. http://www.windowsazure.com/en-us/.
5. Azure drives. http://msdn.microsoft.com/en-us/library/windowsazure/ee924681.aspx.
6. Azure priceing. https://www.windowsazure.com/en-us/pricing/details/.
7. Eucalyptus. http://www.eucalyptus.com/.
8. HDFS. http://hadoop.apache.org/hdfs/.
9. Y. Bu, B. Howe, M. Balazinska, and M. D. Ernst. Haloop: efficient iterative data processing on large clusters. *Proc. VLDB Endow.*, 3:285–296, September 2010.
10. S. P. C. Vecchiola and R. Buyya. High-Performance Cloud Computing: A View of Scientific Applications, Pervasive Systems, Algorithms, and Networksm. In *ISPAN9: 4-16*.
11. P. Carns, W. B. Ligon, R. B. Ross, and R. Thakur. PVFS: a parallel file system for linux clusters. In *In Proc. of the 4th annual Linux Showcase & Conference 2000*.
12. A. Dave, W. Lu, J. Jackson, and R. Barga. Cloudclustering: Toward an iterative data processing pattern on the cloud. In *IPDPSW, 2011*, pages 1132 –1137, may 2011.
13. M. Dorier, G. Antoniu, F. Cappello, M. Snir, and L. Orf. Damaris: How to Efficiently Leverage Multicore Parallelism to Achieve Scalable, Jitter-free I/O. In *Cluster 2012*.
14. J. Ekanayake, H. Li, B. Zhang, T. Gunarathne, S.-H. Bae, J. Qiu, and G. Fox. Twister: a runtime for iterative mapreduce. In *HPDC '10*, pages 810–818, 2010.
15. T. Gunarathne, T.-L. Wu, J. Qiu, and G. Fox. Mapreduce in the clouds for science. In *CloudCom10*, pages 565–572, 2010.
16. T. Gunarathne, B. Zhang, T.-L. Wu, and J. Qiu. Scalable parallel computing on clouds using twister4azure iterative mapreduce. *Future Generation Computer Systems.*, 2012.
17. T. Hey, S. Tansley, and K. M. Tolle, editors. *The Fourth Paradigm: Data-Intensive Scientific Discovery*. Microsoft Research, 2009.
18. T. F. K. Keahey, J. Bresnahan and D. LaBissoniere. Cumulus: Open Source Storage Cloud for Science. In *SC2010 Poster*.
19. Y. Luo and B. Plale. Hierarchical MapReduce Programming Model and Scheduling Algorithms. . 2012.
20. D. Moise, A. Carpen-Amarie, G. Antoniu, and L. Bougé. A Cost-Evaluation of MapReduce Applications in the Cloud. In *In Proc. of the Grid5000 School, Reims, France, 2011*.
21. B. Nicolae, G. Antoniu, L. Bougé, D. Moise, and A. Carpen-Amarie. BlobSeer: Next Generation Data Management for Large Scale Infrastructures. *Journal of Parallel and Distributed Computing*, 2010.
22. K. H. O. Tatebe and N. Soda. Gfarm grid file system. In *New Generation Computing: 257-275*.
23. S. Ostermann, A. Iosup, N. Yigitbasi, R. Prodan, T. Fahringer, and D. H. J. Epema. A Performance Analysis of EC2 Cloud Computing Services for Scientific Computing. In *CloudComp'2009: 115-131*.
24. J. Poline, C. Lalanne, A. Tenenhaus, E. Duchesnay, B. Thirion, and V. Frouin. Imaging genetics: Bio-informatics and bio-statistics challenges. In *COMPSTAT Paris, 1:101-113*.
25. P. Schwan. Lustre: Building a file system for 1000-node clusters. In *Proceedings of the Linux Symposium 2003*.
26. R. Tudoran, A. Costan, and G. Antoniu. MapIterativeReduce: A Framework for Reduction-Intensive Data Processing on Azure Clouds. In *Third International Workshop on MapReduce and its Applications (MAPREDUCE'12), held in conjunction with ACM HPDC'12.*, Delft, Netherlands, 2012.
27. R. Tudoran, A. Costan, G. Antoniu, and L. Bougé. A Performance Evaluation of Azure and Nimbus Clouds for Scientific Applications. In *CloudCP 2012 – 2nd International Workshop on Cloud Computing Platforms, Held in conjunction with the ACM SIGOPS Eurosys 12 conference*, Bern, Switzerland, 2012.
28. R. Tudoran, A. Costan, G. Antoniu, and H. Soncu. TomusBlobs: Towards Communication-Efficient Storage for MapReduce Applications in Azure. In *12th IEEE/ACM International Symposium on Cluster, Cloud and Grid Computing (CCGrid'2012)*, Ottawa, Canada, 2012.
29. M. Zaharia, M. Chowdhury, M. J. Franklin, S. Shenker, and I. Stoica. Spark: cluster computing with working sets. In *HotCloud'10*, pages 10–10. USENIX, 2010.
30. Y. Zhang, Q. Gao, L. Gao, and C. Wang. imapreduce: A distributed computing framework for iterative computation. In *IPDPSW, 2011*, pages 1112 –1121, may 2011.

DOI: 10.1002/cpe