Journal of Internet Services and Applications

RESEARCH Open Access

Upgrading a high performance computing environment for massive data processing

Lucas M. Ponce[1*], Walter dos Santos[1], Wagner Meira Jr.[1], Dorgival Guedes[1], Daniele Lezzi[2] and Rosa M. Badia[2,3]

Abstract
High-performance computing (HPC) and massive data processing (Big Data) are two trends that are beginning to converge. In that process, aspects of hardware architectures, systems support and programming paradigms are being revisited from both perspectives. This paper presents our experience on this path of convergence with the proposal of a framework that addresses some of the programming issues derived from such integration. Our contribution is the development of an integrated environment that integretes (*i*) COMPSs, a programming framework for the development and execution of parallel applications for distributed infrastructures; (*ii*) Lemonade, a data mining and analysis tool; and (*iii*) HDFS, the most widely used distributed file system for Big Data systems. To validate our framework, we used Lemonade to create COMPSs applications that access data through HDFS, and compared them with equivalent applications built with Spark, a popular Big Data framework. The results show that the HDFS integration benefits COMPSs by simplifying data access and by rearranging data transfer, reducing execution time. The integration with Lemonade facilitates COMPSs's use and may help its popularization in the Data Science community, by providing efficient algorithm implementations for experts from the data domain that want to develop applications with a higher level abstraction.

Keywords: COMPSs, High-performance computing, Big data, HDFS, Lemonade

1 Introduction

Parallel and distributed computing frameworks have proven to be essential for applications that require high performance, usually associated with the processing of large volumes of data. Originally, efforts in that area originated from two different areas, High-Performance Computing (HPC) and Big Data. More recently there has been a tendency to combine efforts from both areas to merge their contributions. This work fits in that direction.

HPC applications are those that explore high-level parallelism and high-performance hardware, including low latency networks, to process mostly structured data with scientific algorithms. On the other hand, Big Data scenarios involve the processing of massive data volumes (usually unstructured), leveraging the use of conventional hardware and exploiting data parallelism. In this case, data could be processed as multiple individual streams and analyzed collectively in stream or in batch, for the discovery of knowledge. In such scenarios, data mining in big data has become one of the key tasks in many fields of Science [1].

Considering the convergence of HPC and Big Data, several proposals have emerged to address the requirements of those two areas [2–4]. HPC environments generally provide better interfaces for regular data and scientific algorithms based on bag-of-tasks models such as matrix computation. Despite the good performance in those scenarios, it is often hard to implement applications that handle irregular data and complex data structures in HPC frameworks. Big Data environments offer good solutions to address such kind of data, as well as to facilitate the development of applications by experts in the application

*Correspondence: lucasmsp@dcc.ufmg.br
[1]Departamento de Ciência da Computação, Universidade Federal de Minas Gerais (UFMG), 31270-901 Belo Horizonte, Minas Gerais, Brazil
Full list of author information is available at the end of the article

domain. In this work we aimed at extending an environment commonly used in HPC scenarios, COMPSs, with a distributed file system and a visual development environment for data mining applications, solutions usually associated with Big Data environments. Besides the resulting system in itself, our contributions include a discussion about how abstractions from both areas can benefit each other and how they can be effectively integrated.

COMPSs implements a task-based programming model, a model that has proven to be suitable for HPC applications [5, 6]. It provides a task-based abstraction that is easily understandable by programmers in the HPC community. However, in Big Data scenarios other abstractions have been adopted, like Spark, one of the most widely used Big Data frameworks. In Data Science scenarios, one of the main advantages of Spark is the wide range of available libraries (e.g., MLlib, GrapX, Streaming and SparkSQL, and other integrated tools) [7].

In this context, this paper proposes an extension of COMPSs with two different contributions: first, by adding support for the Hadoop Distributed File System (HDFS), the distributed file system most commonly used for Big Data; second, by integrating it into a massive data-processing application development environment that reduces the user's need to know the details of a programming language to produce applications. Through HDFS integration, we intend to facilitate the use of large volumes of data in COMPSs. In addition, by using a visual development environment, we can hide many details of a parallel programming language, making COMPSs accessible to a larger number of users. In order to achieve that, we adopted the Lemonade environment, a data mining and analysis tool developed at the Universidade Federal de Minas Gerais (UFMG) [8]. The integration of COMPSs with HDFS and the version of Lemonade that outputs COMPSs code are both open-source and are available on GitHub. We also present a performance comparison of COMPSs and Spark applications, using a cluster usually associated with Big Data scenarios, with virtualized nodes and without shared disks.

To describe our work, the remainder of this paper is structured as follows: Section 3 introduces the COMPSs framework; Sections 4 and 5 present its integration with HDFS and Lemonade, respectively. The evaluation of our solution is discussed in Section 6 and finally Section 7 presents our conclusions and discusses future work.

2 Related work

COMPSs is a framework in constant development, it is receiving several new extensions and APIs to fit Big Data requirements, that vary from cloud connectors [9] to a resource manager integration [10]. In a recent work [3], COMPSs Storage API is presented, an official software interface that allows COMPSs applications and COMPSs runtime to work with persistent objects. The Storage API can be deployed on multiple back-ends and it allows the creation, removal, insertion, retrieval, interaction with persistent data, and especially the extraction of the local information about that data. The authors demonstrate its usage by providing an integration with Apache Cassandra [11], a non-relational (NoSQL) database storage.

Currently, besides HDFS, distributed storage systems have become more diverse with a variety of purposes: file systems such as NFS [12], Alluxio [13], Amazon Simple Storage Service (S3) [14], Microsoft Azure Storage [15], Lustre [16]; object stores such as OpenStack Swift [17] and Ceph [18]; key-value systems such as FAWN-KV [19], Dynamo [20] and Memcached [21]; and NoSQL databases such as Apache Cassandra and Apache HBase [22]. Each of these systems address specific problems, even for storage systems of similar categories. Some of them are specialized to handle large volumes of data (e.g., Amazon S3, Cassandra, HBase and HDFS), while others focus on increasing I/O bandwidth (e.g., Alluxio, Lustre and Memcached). Each of the storage systems mentioned has its particularities, was developed with a specific problem in mind, but often can interact with others. For example, Alluxio (formerly Tachyon) provides an efficient in-memory data sharing layer by using existent storage such as HDFS, NFS or S3 as a persistence layer.

Among all the available systems, we decided to integrate COMPSs with HDFS, since it is one of the most widely adopted solution in the market. It supports multiple replicas of files, which increases access bandwidth for multiple clients accessing a single file (a known bottleneck in NFS); however, it is not a good solution for handling small files. Besides that, many of the other solutions mentioned are often implemented on top of it (e.g., HBase, Cassandra and Alluxio). Perhaps the storage system closest to HDFS proposal is Amazon S3, a subsidiary service of Amazon Web Services for cloud storage. S3 aims to provide storage at a low cost, as a highly available service using a price model based on "pay-as-you-go". However, S3 is a proprietary solution and it lacks some functionality often required in scientific projects, such as flexible access control and support for delegation, for example, in large science collaborations groups [23].

The dataflow model is a trend in Big Data applications [24]. There are several flow-based programming frameworks often defining applications as networks of "black box" processes, which exchange data through predefined inputs and outputs. Those frameworks use different ways to define a flow: by using a functional-based language, such as Apache Spark and Twister2 [24]; by using a skeleton-based pipelines, such as Ruffus [25] and Cosmos [26]; or even visually, such as RapidMiner [27], Orange [28] and KNIME [29].

Although code-based frameworks, such as Apache Spark, provide a good degree of parallelism and customization, programming is often complex because needs a high-knowledge from user to understand the syntax and its operators. In Ruffus it is possible to create multi-thread workflows automatically; however, it requires users to encode each function of their pipelines and to explicitly define how task functions are connected and how data will be exchanged. In Cosmos, it is possible to create flows with the MapReduce paradigm [30]; however, besides the code for all functions, it is also necessary to define their dependency graph, which requires a specific syntax.

Visual data flows tools enables users to construct complex data analysis scenarios without programming by supporting a visual interface. Besides the different abstraction levels adopted in each tool to represent a workflow, a common feature is the support of drag, drop and connect operations to work with the available components. Many of them, such as RapidMiner, Orange and KNIME, are designed to be used locally, making them inappropriate to process large data sets that exceed the capabilities of a single machine. ClowdFlows [31] and Orange4WS [32] are popular web-based solutions that support non-local processing, enabling them to execute in a cluster. However, their solution is multithread-based: it allows multiple workflows to execute concurrently by distributing them through nodes, but does not provide distribution or parallelization of data within a flow to handle Big Data scenarios. Lemonade is similar to Microsoft Azure Machine Learning (ML) Studio [33] in that both support creation and execution of applications from a visual interface using Apache Spark to handle Big Data. Azure ML, however, is a proprietary solution that requires a subscription with the Microsoft Azure, being restricted to a cloud architecture. Lemonade is an open-source solution that support both cluster and cloud architectures. Originally, the generated codes were expressed in Spark language [8]; in this work it is extend to also be able to create and execute operations in COMPSs.

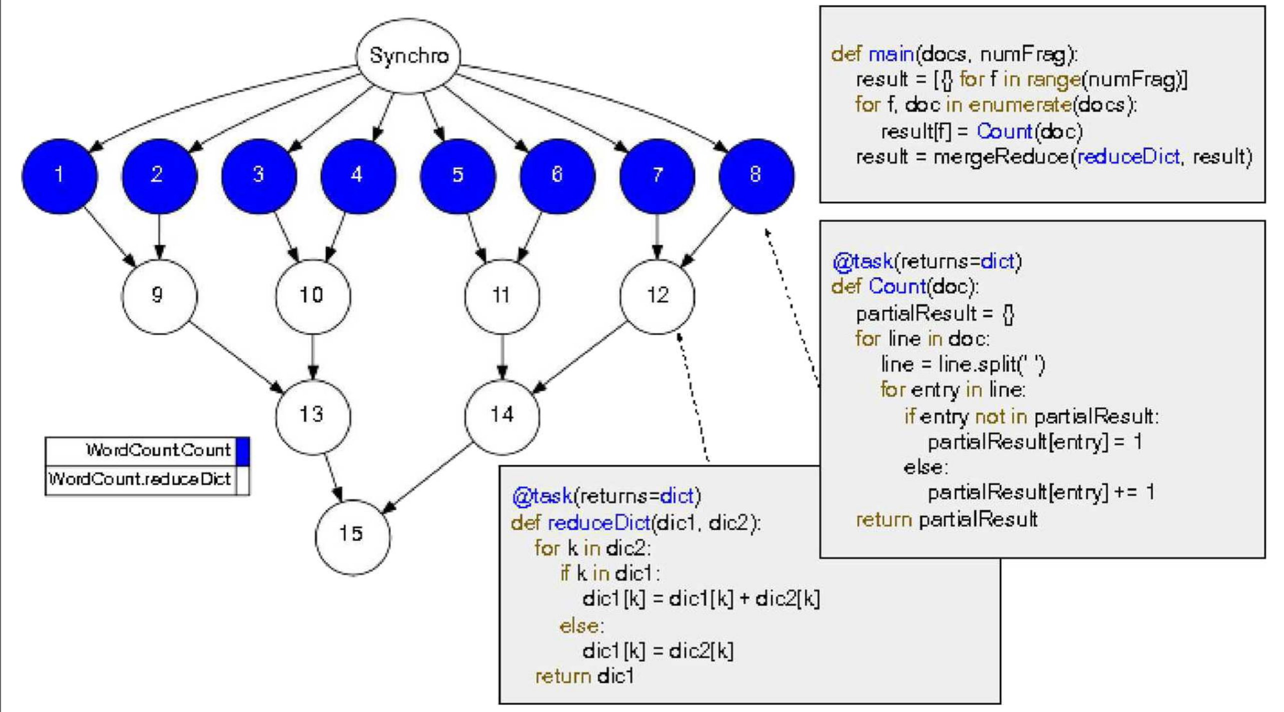
A recent work [34] compared COMPSs performance in Java applications to Apache Spark, using a cluster architecture normally associated with HPC applications (e.g., low-latency networks and shared network disks). In this work, our integration allow us to take COMPSs into a cluster usually adopted in Data Science scenarios, with only traditional networking hardware and with disks distributed among the cluster nodes.

3 The COMPSs framework

COMPSs is a general data processing framework whose main objective is to ease the development of applications for distributed environments, composed of a programming model and an execution runtime that supports it. Applications in COMPSs are written following the sequential paradigm with the addition of annotations in the code that are used to inform that a given method is a task. That means it can be asynchronously offloaded at execution time, and can potentially be executed in parallel with other tasks. In the case of Java and C++, those annotations are provided in an interface file that indicates the directionality of the parameters (input or output). In the case of Python, tasks are identified with an annotation in the form of a decorator started with "`@task`" on top of a method. With that information, the COMPSs runtime generates a task graph at execution time where each node denotes a task, and edges between them represent data dependencies identified based on the tasks's parameters and return values. The task graph expresses the inherent parallelism of the application at task level, which is exploited by the runtime.

Regarding the programming model, to port an application to COMPSs, besides requiring the identification of the functions that are tasks, it may require structural changes to the code in order to improve application efficiency and achieve more parallelism. A very common case is, for example, an application with a single input, possibly a big file, that has to be processed by a task to extract information from it. The first and quick solution would be to assign the entire input file to a task and let it read and compute the data. If there are no dependencies among file data elements, a much more efficient approach in COMPSs, which exploits a higher level of parallelism, is to split the input file into several fragments and invoke multiple tasks, one per fragment. In that way, different resources will be used to execute, in parallel, the different tasks. Figure 1 shows the Python version of a word count application in COMPSs that uses such input fragmentation technique. The dependency graph produced during execution is also shown. Before the integration with HDFS (which will be presented in Section 4), the programmer needed to explicitly split the input file in the desired number of fragments before the task could be called.

The application's idea is to have the input broken into fragments and to count the words in each fragment of the file. Then, it uses a COMPSs operator, `mergeReduce`, to combine all the separate counts as a distributed reduction operation. As shown in Fig. 1, the code in Python does not involve any new syntax, different from other distributed programming frameworks, like Spark. In the example, annotations for both tasks (count words in a fragment and perform partial sums during the reduction) are similar: they indicate that both `Count` and `reduceDict` are parallelizable tasks which return dictionaries as results. The execution graph shows that all `Count` tasks can be executed in parallel and partial reductions also have some parallelism, which was controlled by the `mergeReduce` function.



```
def main(docs, numFrag):
    result = [{} for f in range(numFrag)]
    for f, doc in enumerate(docs):
        result[f] = Count(doc)
    result = mergeReduce(reduceDict, result)
```

```
@task(returns=dict)
def Count(doc):
    partialResult = {}
    for line in doc:
        line = line.split(' ')
        for entry in line:
            if entry not in partialResult:
                partialResult[entry] = 1
            else:
                partialResult[entry] += 1
    return partialResult
```

```
@task(returns=dict)
def reduceDict(dic1, dic2):
    for k in dic2:
        if k in dic1:
            dic1[k] = dic1[k] + dic2[k]
        else:
            dic1[k] = dic2[k]
    return dic1
```

Fig. 1 Wordcount application in PyCOMPSs. The topmost code block shows the main function, that triggers the calls to Count and reduceDict, the two functions defined as tasks, shown in the two other code blocks. The graph shows the tasks created during execution and the dependencies among them, identified based on the parameters and return values of each task

The COMPSs runtime architecture is based on a main component, the master that executes the main code of the application, and a set of worker processes deployed on computational nodes that execute the tasks. Those nodes can be part of a physical cluster, dynamically instantiated virtual machines, or containers. The runtime takes care of data transfers, task scheduling and infrastructure management. It relies on an interoperability layer that makes COMPSs able to communicate with several resource managers.

When tasks need to read files stored in a conventional file system without a shared disk, those files need to be available in the central node (which is executing the `main` code). The document fragments are read in the `main` function, and passed as parameters to the `Count` tasks. Then, COMPSs runtime would be responsible for transferring the data over the network to each working node that would execute one of the tasks, which implies an overhead on the access to the data. The integration with HDFS presented in this work simplifies the split of the input data and distribution of the blocks to worker nodes, leveraging the COMPSs runtime to use data locality to make better scheduling decisions.

4 COMPSs-HDFS integration

The HDFS file system, distributed under the Hadoop project, was developed to deal with the partioning, distribution and access of massive file shares with sequential data access patterns, running on clusters of commodity hardware ([35], p. 43). In HDFS, each file is internally divided into blocks (usually, 64 MB or 128 MB in size), which are automatically distributed among the storage nodes (datanodes). To achieve fault tolerance, increase data availability and access bandwidth, each block can be replicated on multiple nodes.

When a client needs to access a file, HDFS provides a list identifying all the blocks that compose the file to the client with information about their locations and replicas. Using that information, the application can decide the proper way to distribute its blocks (i.e., parts of the file) among the processing nodes (workers). Each worker receives a number of blocks to process and can access HDFS datanodes directly to retrieve, in parallel, the content of such blocks. When a worker needs to access data, it can fetch them from the best source based on location and datanode load. In cases where the data is hosted on the same compute node, HDFS clients are able to access such data directly through a Short-Circuit ([35], p. 308).

The main concept in the proposed integration between HDFS and COMPSs is the delegation of some responsibilities to HDFS, such as the division of the input files in blocks and the transfer of those blocks to each COMPSs worker. The first step was to decide how HDFS abstractions should be made available to the COMPSs programmer.

4.1 Data abstraction

The integration presented here[1] provides APIs in Python and Java. We chose those two languages because Java is the native language for COMPSs and HDFS, while Python popular in Data Science scenarioes, and it is required for the Lemonade environment, which will be described later. Each API provides two abstractions with well-defined functions for the COMPSs programmer. The first abstraction, represented by the *HDFS* class, is responsible for dealing with HDFS directly; for instance, to create folders or to retrieve information about a file. The second, represented by the *Block* class, is responsible for the representation of files divided in fragments, which includes methods like `readBlock` (reads a fragment as a string buffer, which can be read as a common file), `readDataframe` (reads a fragment as a DataFrame, a method used in Lemonade) and `readBinary` (reads a fragment as binary data).

The idea is that, for reading, the programmer will first use the API to retrieve information about the list of fragments of a given file in HDFS. After that, each element of this list will be sent to a worker, which will retch its data. There are two slightly different interfaces that the programmer can use to retrieve the fragment list. In one, the user can request the target file to be represented as a list of exactly n fragments, e.g., matching the number of cores available in the cluster. In the other, the user requests the information about the file as the exact list of blocks that HDFS used to split it during its creation. In this case, we use the COMPSs Storage API extension, discussed in Section 2, to schedule the tasks on the workers that own each HDFS block to be processed. In both cases, when a COMPSs task reads data, the Block entity will choose, through HDFS, the best provider (Datanode) for each fragment. However, when using the later API, we have a greater chance of activate Short-Circuits to read a block, because COMPSs can access the block location from the list while scheduling tasks.

Algorithm 1 illustrates the basic procedure to use the HDFS integration in COMPSs. *BLOCKSLIST*, in the example, does not contain the HDFS file itself; each block in the loop is a light reference that contains, for example, the offset of the initial byte that marks the beginning of the block. Inside a task, a worker will use the information about its fragment to request the data from HDFS, which will, in turn, coordinate the transfers. Using HDFS, each fragment can be read in parallel by the multiple instances of the task (*task1*). From there, the next steps are similar to the existing solutions in COMPSs programming, that

Algorithm 1: COMPSs HDFS API usage example.

```
begin
    BLOCKSLIST = retrieves a list with fragments
    from a file on HDFS;
    for BLOCK_INFO ∈ BLOCKSLIST do
        task1(BLOCK_INFO);
    end for
end
```

is, each partial result can be saved to a separate file or can be used as input to a new task.

As previously mentioned, HDFS delimits blocks by number of bytes (physical blocks). However, processing each block in this way might not be practical, because most operations on data impose a logical interpretation of records, delimited by language- or application-specific markers, leading to variable-length records (e.g., records represented as text lines). Even when applications handle fixed-length records, if those have a size that is not the same as the HDFS-defined file block size, their boundaries would not match those defined by HDFS. So, if methods like `readBlock` and `readDataframe` considered only the amount of bytes to read, we could process data improperly. Figure 2 illustrates the difference between logical blocks and physical blocks for a 350 MB file where each of the seven records (lines) contains 50 MB. When a task is assigned to process the first HDFS block, it has access to three lines, but the bytes that area at the end of *Line 3* will have to be requested for the Datanodes that hold *Block 2*. Similarly, the worker that processes *Block 2* will have to request the last bytes of *Line 6* from the datanode holding *Block 3*; in this case, it will also to discard the first bytes of the block, which actually represent part of the content of *Line 3*.

To solve this problem, those two methods check whether the block is the first or the last of the file. If the block is not the last, the method requests consecutive portions of 2 KB data from the next block until it finds a record delimiter (e.g., a newline). In addition, if the block is not the first, the method skips the first bytes until the end of the first record/line. In this way, we guarantee that the blocks of each task will maintain their logical meaning.

Figure 3 shows the resulting Wordcount when the HDFS API is used. Conceptually, the operation is the same, but now we handle the file through HDFS; the different lines in relation to Fig. 1 are marked by red braces. We can see, in the `main` method, the command to contact HDFS and to request the information about a specific file. In the `Count` method, each fragment is read as text, similar to a conventional file. All tasks can read data from their particular file block, in parallel, which improves

[1] https://github.com/eubr-bigsea/compss-hdfs

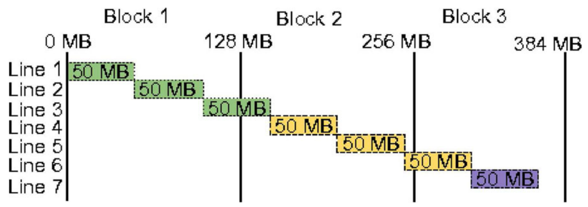


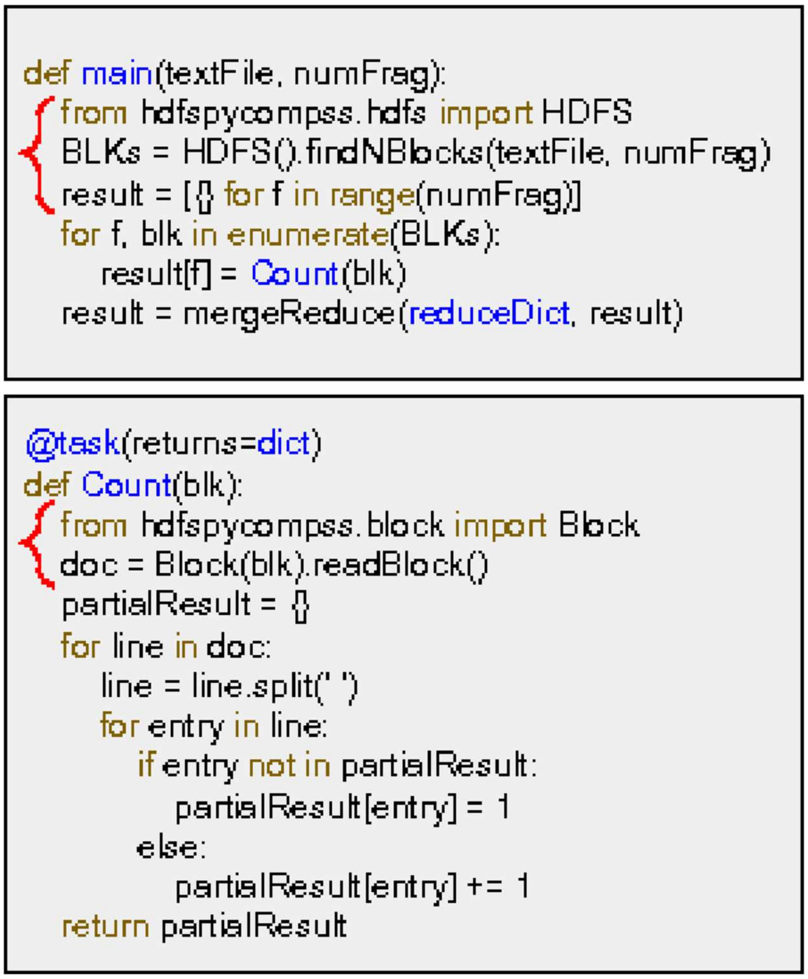
Fig. 2 Difference between logical and physical blocks due to HDFS fixed partition boundaries

data transfer times. The `reduceDict` method is omitted because there is no difference compared to Fig. 1.

When writing to a file, a merged output can be written into HDFS by the master node if it fits in its memory (so it can be collected from the workers). Otherwise, each task can create a partial file in HDFS with its part of the output. If the user wants, those partial files can then be merged using the API. An append function, which would allow every task to write its output to the end of a global file, is not provided, since it would impose a serialization in the execution, given that HDFS does not allow concurrent writes to a single file.

4.2 Communication with HDFS

To integrate COMPSs with HDFS, we considered the aspects of techniques available to implement the communication between external applications, in particular those written in Java and Python, and HDFS. HDFS provides



```
def main(textFile, numFrag):
    from hdfspycompss.hdfs import HDFS
    BLKs = HDFS().findNBlocks(textFile, numFrag)
    result = [{} for f in range(numFrag)]
    for f, blk in enumerate(BLKs):
        result[f] = Count(blk)
    result = mergeReduce(reduceDict, result)
```

```
@task(returns=dict)
def Count(blk):
    from hdfspycompss.block import Block
    doc = Block(blk).readBlock()
    partialResult = {}
    for line in doc:
        line = line.split(' ')
        for entry in line:
            if entry not in partialResult:
                partialResult[entry] = 1
            else:
                partialResult[entry] += 1
    return partialResult
```

Fig. 3 Wordcount application code in PyCOMPSs using the HDFS extension. Changes from the previous version are marked by red braces, where the code now uses the HDFS API

interfaces through a direct Java API, a shell command line (CLI), a REST API (webHDFS) and a C API (libhdfs). Because HDFS is written in Java, the most complete interface and the most powerful one is the Java API; the others implement many aspects of the service, but do not guarantee a perfect correspondence to the Java API. While in the communication between Java-COMPSs applications and HDFS it makes sense use the HDFS Java API interface, we had to consider some alternatives for the communication in Python, because HDFS does not offer native support in that language. We sought a solution capable of handling large data transfers, with access to low-level features, such as to open a file from a given position (a byte offset), a feature that was required in our HDFS data abstraction.

WebHDFS is a native HDFS API for communication using HTTP REST ([35], p. 54). Although it has been used in many Python modules as an alternative method to handle HDFS files [36–38], its HTTP interface is significantly slower than the native Java client, due both to request timing and the overhead to use the HTTP protocol, so should be avoided for large data transfers. A second approach would be to create a communication service between the Python application and a Java process, similar to what is done by PySpark using the Py4J module [39]. PySpark, which has a hybrid Python and JVM framework, uses Py4J internally only as a request driver for the Spark process in the JVM. In the case of COMPSs, changes in its source code would be needed for that to work and it would not be possible to maintain such a level of integration over version updates. Besides, each thread created in COMPSs would have to communicate with a Java interleaver (which would in turn connect to HDFS) externally to COMPSs to request and receive data. In addition to such external processes increasing memory consumption, data transfers would have to go through the intermediary itself, different from what occurs in PySpark. LibHDFS ([35], p. 55) is the most used approach in Python modules [40, 41]; is uses a Java wrapper in C that communicates with Java using the Java Native Interface (JNI), a technology for communicating applications directly in the Java JVM. Besides the existence popular systems using that solution, such as PyArrow [41], it provides a high-level file abstraction, not giving access to some low-level API features required by our project.

Considering those limitations, our chosen solution uses libhdfs3, an alternative implementation of the libhdfs protocol developed by the Apache HAWQ project [42]. While the original libhdfs uses JNI, the libhdfs3 uses a Hadoop RPC protocol. This difference gets rid of the drawbacks of JNI, provides a lightweight, small memory footprint code base, and is able to exploit features such as Short-Circuit. In turn, Python has, by default, modules capable of interpreting and converting C/C++ language data types. Based on that, it was possible to use libhdfs3 to create a mapping of the functions and data types to be used in the HDFS integration. Because Python invokes methods in C++ that are run internally, this approach is faster and more efficient than webHDFS or the Py4J module.

5 Lemonade

In the big data area, the application domain experts responsible for analyzing the data often are not computer scientists, and usually lack any experience in parallel/distributed programming. A recognized challenge for those researchers is to express their queries in a processing tool. Although COMPSs reduces the demand in terms of parallel programming, it still requires the developer to identify the tasks that can be run in parallel and to program them in a language like Java or Python. To reduce those barriers, we decided to integrate COMPSs with Lemonade, a visual big data programming environment. As mentioned in Section 2, there are several tools that support visual data flows, such as RapidMiner, Orange and KNIME. However, those are designed to be used locally, making them inappropriate to process large data sets that exceed the capabilities of a single machine. Although other platforms, such as Microsoft Azure Machine Learning (ML) Studio and ClowdFlows support non-local processing, enabling them to use a cluster still presents several challenges.

Lemonade[2] is a visual tool designed for data scientists, targeting users who lack programming skills or who want to develop workflows using the existing modules of that tool [8]. The platform focuses on the creation of analysis and mining flows in the cloud or on a private cluster, with authentication, authorization and access accounting guarantees. Using an interface for visual construction of flows, it allows users to choose predefined operations, drag and connect them to compose and execute flows by encapsulating the details of storage, coding, security and distributed processing, allowing them to be used in cloud environments by data domain experts. Figure 4 is an example of an application created with Lemonade, in that case, a classification application using KNN. Each operation is presented as a box that represents a data manipulation task, for instance, a machine learning algorithm. Each box may have a set of parameters that must be specified to control its execution like, for example, the name of a file to be read, or the maximum number of iterations for an algorithm.

In its original version, Lemonade had guidelines to generate Spark 2.0.2 code in Python (PySpark). Once integrated with COMPSs, it can now offer its users algorithms written with COMPSs, also in Python (PyCOMPSs). Such

[2]https://github.com/eubr-bigsea/docker-lemonade

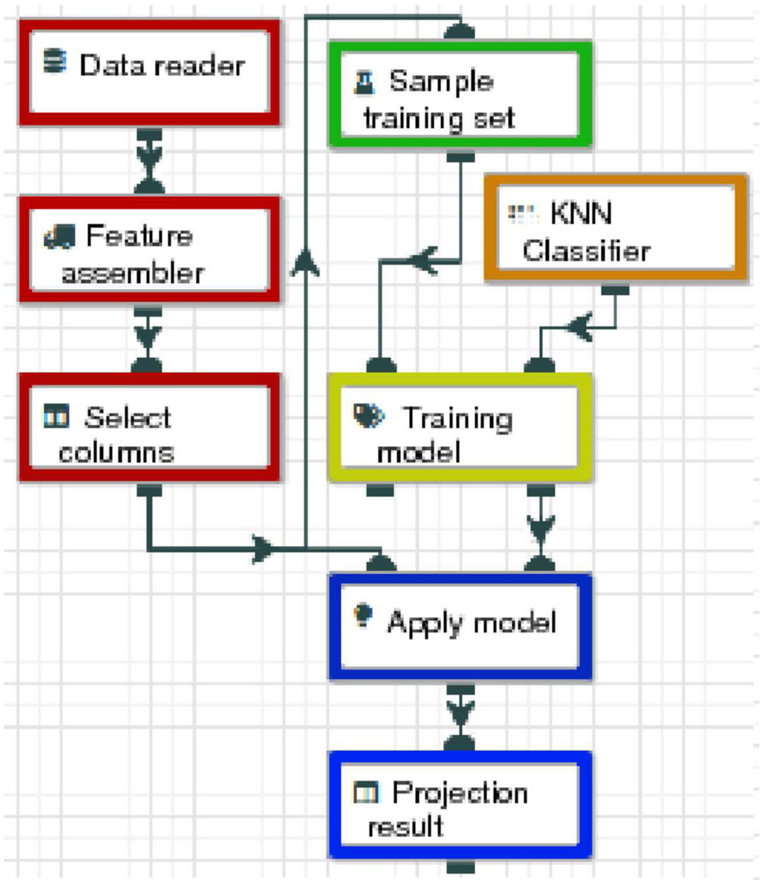


Fig. 4 KNN workflow for a KNN classification application created using Lemonade. The data reader extract data from a file; specific features are extracted from records and then only the columns necessary for classification are selected. From that data, a sample is taken to feed the training model, which uses the KNN classifier as its engine. The trained model is then applied to the remainder of the data, and the Projection box provides a visualization of the result. The colors are used to identify modules that can be grouped during code generation (discussed in Section 5.3)

algorithms can then be combined to build complex workflows. Thus, this strategy enables a hierarchical composition of code and promotes code reutilization.

5.1 The lemonade environment

The Lemonade architecture is composed of seven individual components that work as micro services, responsible for the web interface (Citron), security and privacy (Thorn), workflow execution (Juicer), application monitoring (Stand), management of data sets meta-data (Limonero), algorithms meta-data (Tahiti) and output data visualization (Caipirinha).

To integrate COMPSs with Lemonade, operations and algorithms reflecting the modules already available in its Spark version were implemented in COMPSs and registered in Tahiti, which maintains all information about the available functions. Such meta-data includes, for instance, the category of each operation (e.g., text operations or machine learning algorithms) and their parameters (e.g., column names or the maximum iteration for a given algorithm). The Juicer module was extended, since it is responsible for translating the workflow created by the user (stored as a *JSON* file) into source code, which it then submits for execution in a cluster allocated for that. We created a new source-to-source compiler (transpiler) that reads the *JSON* file and generates COMPSs code. This code consists of two parts, one that is dynamically generated, which includes a main method responsible for coordinating all the calls to the operations used in the application, and another that is a library with the implementations of operations and algorithms registered in Tahiti.

5.2 Algorithms and operations

While Spark extracts parallelism from functional operators, COMPSs does it by identifying the lack of dependencies between tasks. The idea behind the implementation of a COMPSs module in Lemonade is to take advantage of the notion of HDFS blocks and to decompose large data files into pieces, so that the algorithm can be broken into smaller tasks, each one processing a data block with little or no dependencies to other tasks, so they can be executed in parallel.

The choice of algorithms to be implemented using COMPSs in Lemonade was based on the ones that had already been made available using Spark. So far, 44 functions have been implemented with COMPSs, shown in Table 1. They can be divided into seven major categories: read/write operations and change of data structure (Data); transformation and information extraction operations (ETL); machine learning algorithms (ML); operations on textual data (Text); quality appraisers of the machine learning models (Metrics); operations and algorithms on geo-referenced data (Geographic); and algorithms on graphs (Graph). All the code is available and documented on GitHub[3].

To ensure the compatibility of modules inputs and outputs during code linkage, all functions have a standard interface with a single input data element and a configuration dictionary as parameters. DataFrame is the data abstraction used in all algorithms and operations. Internally, the data input is a list of Pandas DataFrames [43] where each element is related to an HDFS fragment. The configuration dictionary stores all attributes specified by the user through the interface and input/output data is always a list of *DataFrames* (block concept). Internally to the module, the function can be organized as the programmer wishes, using other functions with any number of parameters.

5.3 Code optimization

By its nature, all communication from one task that outputs some data to another one that consumes that data (as a function input parameter) is mediated by the COMPSs main program. That adds overhead to the execution, that can be avoided in many cases, by optimizing the generated code to avoid that mediation when the intended communication pattern is clearly identified.

The transpiler was implemented in Juicer with initial optimization code guidelines that are based on joining tasks from multiple algorithms and operations into a single task to reduce inter-task communication. Apache Spark implements such optimization internally; each resulting group of operations is called a stage. Such process minimizes the cost of scheduling, data transfer over the network, and creation/removal of an environment for the COMPSs tasks. For instance, in the KNN workflow shown in Fig. 4, the red color operations (*Data reader*, *Feature assembler* and *Select columns*) could be grouped as a single stage. In this case, COMPSs will create only

[3] https://github.com/eubr-bigsea/Compss-Python

Table 1 Operations and Algorithms implemented in COMPSs available in Lemonade

Categories	Operations and algorithms
Data	Read and write files, attributes changer, data balancer
ETL	Add columns, aggregation, clean missing data, difference, distinct (remove duplicate rows), drop columns, filter, intersection, joins (inner, left and right join), replace values, sample, select columns (projection), sort, split, transformation, union.
Geographic	Read shapefile, Geo within (check if a point is within a region), ST-DBSCAN
Graph	PageRank
Metrics	Classification (accuracy, precision/recall and f-measure), regression (MSE, RMSE, MAE, R^2)
ML	Feature assembler, Scalers (min-max, max-abs and standard), String Indexer, PCA, K-Means, DBSCAN, KNN, Naive Bayes, SVM, Logistic regression, Linear regression, Apriori, Load/Save model
Text	Vectorization by Bag-of-Words and Tf-idf, tokenizer, stop-words remover

n tasks (one for each fragment) instead of $3n$ tasks. The same happens with the blue color boxes, *Apply model* and *Projection results*.

In order for this optimization to work, we look for operations that only use local data to execute. Code from a sequence of tasks that only handle local data can be safely integrated into a single stage. In many cases, complex operations may require internal communication before an output can be generated (e.g., when an average over all elements has to be computed). Every time such a general communication pattern is found, integrated stages have to be limited to the tasks before and after it; there can be no direct integration of operations before and after the communication step.

In our implementation of COMPSs modules, all algorithms and operations added to Lemonade were tagged to identify how they manipulate the data from input to output: (*i*) operations that have only one internal step, like *Filter* (in this case, there is no stage of communication between fragments); (*ii*) operations where the number of rows is preserved at the end of execution, such as when some value in a column has to be replaced (mapping); (*iii*) operations that have more than one input (and therefore require communication with more than one box before them, like *Join*); (*iv*) operations that define or change the nature of the data from input to output, like *Split* (which will usually include at least one step where general communication is needed); and, finally, (*v*) operations that write data to HDFS or return data to the main program, like *Save data*.

Based on those tags, the transpiler can decide how to combine (or not) the code of multiple tasks: operations with tags (*iii*) or (*iv*) must always be at the beginning of a stage, since the communication patters inside or just before them preclude the integration of tasks before and after them; operations with tags (*i*) and (*ii*) can be anywhere in a stage, since their operations can be safely integrated; and operations with tag (*v*), by definition, must be at the end of a stage. When those rules lead to the identification of sequences of operations that can be grouped in a single stage, the transpiler outputs code that include all transformations in a single task.

6 Performance evaluation

The main purpose of this work was validate COMPS integration with HDFS and Lemonade. In particular, we sought to better understand the factors involved in the performance of the HDFS integration API and the effects of using Lemonade's code optimization guidelines. In addition, we intended to compare the performance of COMPS applications in Python, created by Lemonade, with Spark applications.

In order to achieve that purpose we used three experimental techniques: *2kr* factorial design, Z-pairwise test and linear regression. A *2kr* factorial design is used to determine the effect of *k* factors, each of which has two alternatives or levels [44]. We used this technique to study the effect of three parameters involved in the use of the HDFS integration API: the interface, the HDFS replication factor and the size of HDFS blocks. The Z-Pairwise test is a technique for comparisons between two systems based on a set of samples using the z coefficient, allowing the calculation of the difference between these systems [44]. In order to satisfy the premise of using the normal coefficient Z, each experiment was executed 40 times. Linear regression was used to evaluate the performance of the executions due to the increase of the workload. In most cases, we have used the one-sided confidence interval when the goal was to show the superiority of a system, but when explicit, we have used the two-side confidence interval to show that there is no significant difference between systems.

We used four applications in the experiments: Grep, an application for occurrence counting of a particular word in parallel; Wordcount, an application of word count also in parallel, where initially it is made a partial count of the words in each fragment, followed by a reduction step for merging results; KNN workflow (shown in Fig. 4), a flow of operations that comprises a step for reading and preprocessing data, a step for sampling data to be used in the training of a K Nearest Neighbors classifier, followed by a final step to apply the model over the data; and a KMeans workflow (shown in Fig. 5), other flow that comprises reading data, creating of a vector of attributes, removing unnecessary columns and training a model to find the centroids of the input data set.

The use cases have different input data types: for Grep and Wordcount, the applications receive a text file; KMeans and KNN workflows receive a tabular file (*csv*)

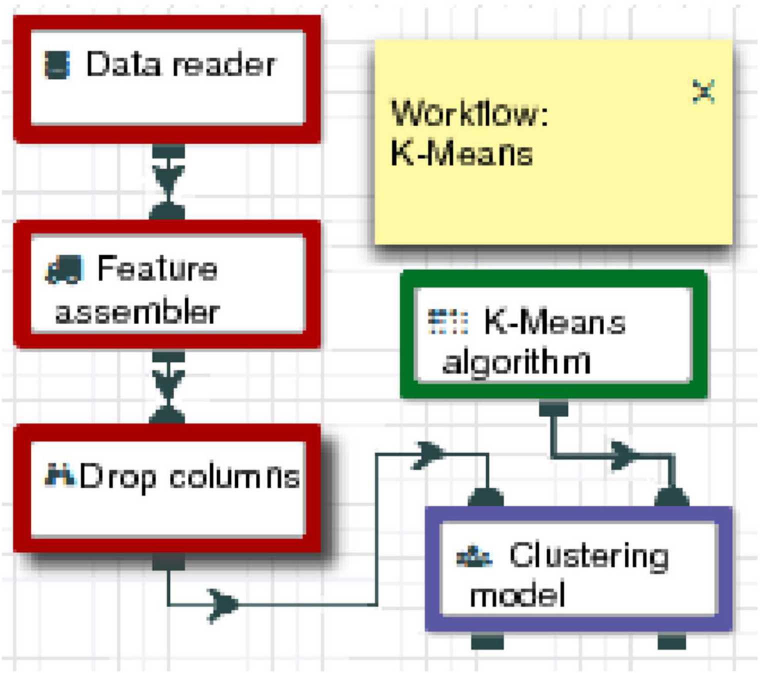


Fig. 5 KMeans workflow created using Lemonade to build a clustering model

with numeric attributes. The text load was created by concatenating several books from the Gutenberg Project[4] multiple times. As for the numerical load, we used the Higgs Data Set[5] which initially contains 11 $\times 10^6$ samples with 28 dimensions of simulated signal processes that produce Higgs Bosons. All experiments used a cluster with COMPSs (v. 2.3), HDFS (v. 2.7) and a Spark (v. 2.2), with a dedicated master node and eight workers nodes. The virtualized machines had Intel E56xx processors of 2.5 GHz with 4 cores, 8 GB of RAM, with Ubuntu Linux 16.04 LTS.

6.1 HDFS read performance

The use of the COMPSs HDFS API involves some parameters that can influence the performance of an execution, being important to specify the used interface, i.e., whether or not it uses the COMPSs Storage API (parameter A), the replication factor (parameter B) and the block size (parameter C). It is expected that the execution be faster when using the Storage API, since it directs, whenever possible, the reading of a fragment to a node that owns it, meaning, it improves data locality. The same happens with data replication, where the usage of a replication factor greater than one increases the probability that a block will be found where the code will execute. Finally, larger block sizes decrease the number of requests made to HDFS, but on the other hand, it can increase the amount of transferred data if the block is in another node and, in addition, it increases memory consumption.

As a first step, we performed a *2kr* factorial Project, shown in Table 2, where *k* corresponds to three evaluated parameters, and *r* to the 40 repetitions for each configuration. For this, we executed scenarios using Grep and Wordcount applications in Python with both interfaces, replications of 1 and 3, and block size of 64 MB and 128 MB.

The result of the factorial project, shown in Table 2, was able to explain approximately 93% of the variation of the execution time for the Grep application; the other 7% may be assigned to experimental errors, like network problems. This analysis shows that the Storage API is the parameter that most impacts the variation of execution time, being responsible for approximately 57% of the variation, where the negative effect means a decreasing of the execution time when it is used. Data replication is the second most impacting parameter that assists in decreasing runtime. These results are expected, since both parameters increase data locality. When using the Storage API, we have a greater probability that the data will be directed to right node. The increase in block size, on the other hand, causes an increase in execution time, although not as significant as the previous ones.

[4]http://www.gutenberg.org
[5]Available in: https://archive.ics.uci.edu/ml/datasets/HIGGS

Table 2 Results of the $2^3 40$ factorial design to analyze configuration aspects of the COMPSs HDFS API

Parameter	Effect	Variation (%)	95% Confidence Interval (two-sided)
Grep			
Intercept	57.97	-	(57.55, 58.39)
A	-11.16	56.93	(-11.58, -10.74)
B	-6.85	21.41	(-7.26, -6.438)
C	4.88	10.9	(4.46, 5.30)
AB	0.51	0.12	(0.09, 0.93)
AC	1.31	0.79	(0.89, 1.73)
BC	-2.2	2.21	(-2.61, -1.78)
ABC	-1.55	1.11	(-1.97, -1.13)
Effects (%)		93.48	
Wordcount			
Intercept	220.57	-	(220.27, 220.87)
A	-7.83	82.44	(-8.13, -7.54)
B	-1.73	4.02	(-2.02, -1.43)
C	-0.35	0.17	(-0.65, -0.06)
AB	0.58	0.45	(0.28, 0.87)
AC	0.40	0.22	(0.11, 0.70)
BC	1.40	2.73	(1.12, 1.72)
ABC	0.56	0.42	(0.26, 0.85)
Effects (%)		90.46	

From Table 2, we can also see the variations caused by the interactions between the parameters, however, since the sum of the interactions represent only about 4% of the variation, we focus only on the individual factors. Since the zero value is not contained in any of the 95% confidence intervals, we can say that all calculated effects, although they come from a sample, are significant, and they represent well the reality given the observed confidence.

The results for Wordcount were similar, where the Storage API and data replication are responsible for most of the variation in runtime, however, with a different ratio of 82% and 4% respectively. Unlike Grep, increasing the size of the blocks decreases execution time, however in a small rate of 0.17%. The results suggest that the decrease in execution time of Wordcount is related to the number of reduction tasks and not to the HDFS itself because, unlike Grep, Wordcount has a more involved cost in the execution of the sum stage of the partial counts. When we use a small block size, in the Wordcount case, we increase the number of tasks for partial reads and, consequently, we increase the number of tasks to join the results. Again, the confidence interval suggests that the effects of the calculated parameters are significant and represent well the population.

After analyzing the importance of the considered parameters, the next step was the creation of speedup[6] graphs, shown in Fig. 6, evaluating the speedup of the different configurations of the HDFS API compared to the conventional file system as a function of the workload. The applications, for both languages, were executed using the conventional file system and both HDFS API interfaces, varying the input load size from 2 to 20 GB, in 2 GB steps and 15 times each. For the executions using HDFS, we also vary the replication factor by 1 and 3 and set the block size to 64 MB. We adopt the following pattern to refer to a COMPSs application: F represents the application that uses the conventional file system; H1 and H3 are versions using the first HDFS interface with replication 1 and 3 respectively; finally, S1 and S3 are versions using the second interface of the API with replication 1 and 3 respectively;

In Fig. 6, we can see that the use of HDFS is preferable in all scenarios because the speedup is always greater than one. The speedup is greater for Grep, as expected, because the Wordcount application has several secondary tasks for the sum of the partial counts. There is a significant difference in runtime between S1 and H1 for simpler applications like Grep, however, the speedups are closer for Wordcount scenarios. For example, using S1 over H1, in Python, increases only 7% in speedup but 1% in Java. For the evaluated workload range, the speedup curves start to stabilize in 14 GB, and at 20 GB, the speedups is approximately 6.9 and 13.9 for the Python Grep application for H1 and S1, respectively. Although the speedup in Wordcount is lower than in Grep, it is still significant. For example, the average Wordcount application runtime in Python with a conventional file system is 577 s, while using H1 or S1 the time is 198 and 184 s, respectively.

There are two reasons for the difference of speedups between Python and Java: the COMPSs architecture and the difference between implementations of Wordcount application. PyCOMPSs is a COMPSs binding: all COMPSs orchestration is done in the JVM while task execution is done in Python. For instance, when transferring files through COMPSs in Python, those files are transparently transferred by connectors executed in the JVM, to be later interpreted by Python. This causes a greater overhead compared to an execution in COMPSs native language. In contrast, when using HDFS, the file is transferred directly by HDFS using libhdfs3 (as discussed in Section 4), minimizing COMPSs overheads. The second reason is the different implementation of the merge step reduction of the partial results. While in Python the results are reduced two by two, in Java, the partial results are reduced in a specific order, one by one, which reduces the time gains during data reading.

In summary, the previous analysis shows that COMPSs HDFS integration increases the performance of applications. Even in more complex applications, where data read step correspond to a smaller portion of the execution, the performance increase of HDFS is at least 50% faster than the conventional file system.

6.2 HDFS write performance

Besides the read performance, it is still necessary to analyze the behavior of our proposed system during data writes. For this, we execute a micro-benchmark application for data writing, varying the use of the file system, HDFS replication and output file size. The application in question is a dumps file creator where each task is responsible for creating a file fragment.

In order to create a COMPSs application using the conventional file system, where each task produces a file, it is necessary tell COMPSs that this file is an output parameter. This means that at the end of the execution, the master node must request the files produced by each task. By default, files and tasks are transparently transferred over NIO connectors. Although this connector is recommended for its speed, its disadvantage is that at the end of execution, COMPSs request all files at once, and that exhausts the central node memory in cases where the total size of the files is larger than the available memory in the master node. For instance, in our cluster, where the master node has 8 GB of RAM, when we try to receive 6 GB of output, COMPSs gives an error due to lack of memory. To work around this problem, there are two alternatives: use the GAT connector, that uses ssh to schedule transfers, or force the serialization of actions when receiving the files. When using HDFS, we are not susceptible to such problems because, as previously mentioned, COMPSs no longer has the responsibility of transferring files between the nodes: that responsibility is now of HDFS.

Figure 7 shows the speedup of the solutions using HDFS with a replication factor 1 (H1) and 3 (H3) over the conventional system using the GAT connector and using the NIO connector with the serialization of the transfers. For this set of tests, we varied the size of the output file from 6 to 16 GB, running each experiment 15 times. The performance of the interface using the Storage API was not evaluated because the API only impacts data reading. Although we only display the results for Python, the results for Java were similar.

We can see from Fig. 7 that the HDFS API is superior in almost all scenarios. Increasing the replication factor, we are writing three times more data, and that affects performance. For instance, writing a 16 GB file using HDFS replication factor 1 (H1) has a speedup of approximately 6x over the conventional system using NIO connectors

[6]A metric that measures the relative performance of two systems processing the same problem.

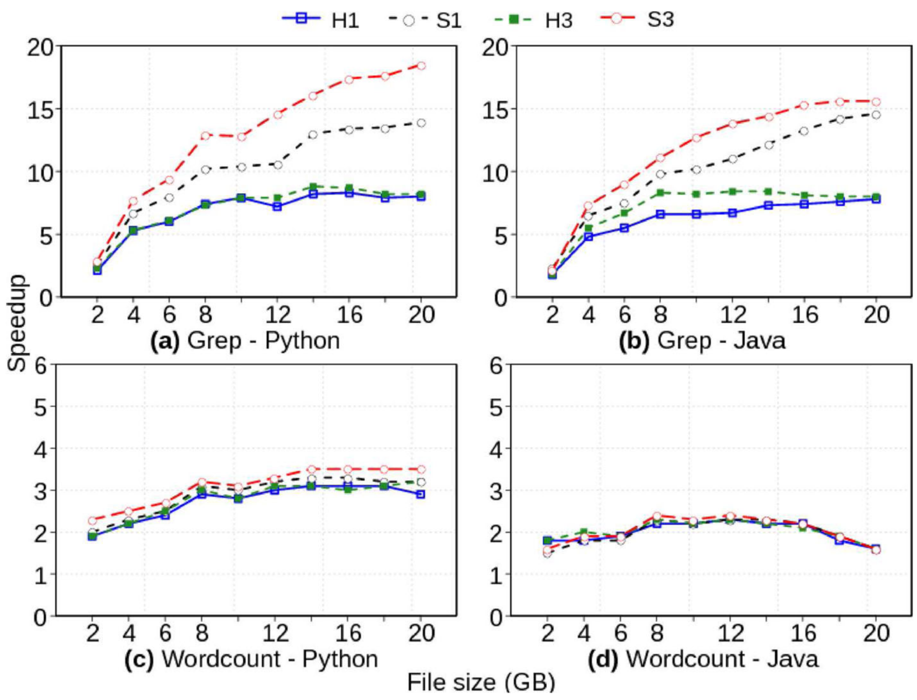


Fig. 6 Speedup as a function of the workload size for Grep and Wordcount applications, considering Python and Java implementations, when the HDFS API is configured with (S) or without (H) the Storage API, and with no block replication (1) or 3-way replication (3). Speedup is computed compared to the execution times of the same applications executing on a conventional file system. **a** Grep - Python. **b** Grep - Java. **c** Wordcount - Python. **d** Wordcount - Java

(serializing file reception). However, the same scenario has a 2x speedup with H3, because internally HDFS was writing 48 GB. In other words, H1 is approximately 3x faster than H3. In that application, it is better to use GAT connector if is necessary to use the conventional file system.

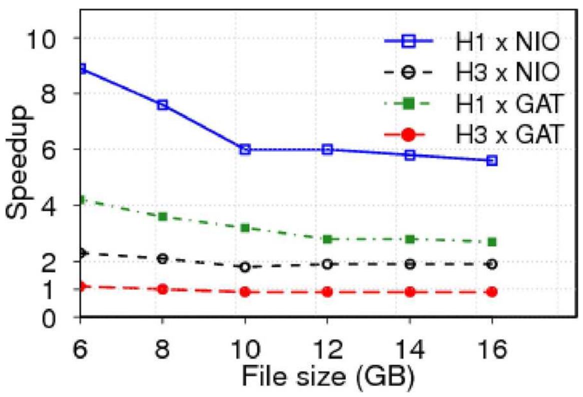


Fig. 7 Speedup of file writing using HDFS over the conventional file system, considering no replication (H1) and 3-way replication (H3), against the performance of the traditional file system interfaces using GAT and NIO solutions

The H1 and H3 speedups over the conventional file system version using the GAT connector is 2.7 and 0.9, respectively. Although the speedup of 0.9 means that writing large amounts of data in HDFS with replication factor 3 (H3) is slower than the conventional system using GAT, we believe that the use of the integration API is still preferable for the writing step. The GAT connector has a larger overhead as disadvantage in COMPS executions. For instance, to this same application when creating a 4 GB file, the runtime was on average 142 s using GAT, however, using NIO connector, without serializing the transfer, the average time was 77 s. Considering that a real application will need to read data, processes tasks and write data, the use of GAT will be slower than executing COMPSs with HDFS, which will use the NIO in all other orchestrations steps.

6.3 Network behavior

In addition to the performance analysis considering execution times, we also evaluated the network traffic to better understand the performance gains of using HDFS in a COMPSs execution. We ran the Grep application for an input load of 20 GB and captured the traffic using Tcpdump[7] in each node of the cluster. We illustrate,

[7]http://www.tcpdump.org/

in Fig. 8, the data exchanges between machines during COMPS executions using the conventional system, using HDFS, and using HDFS while exploring data locality with COMPSs Storage API.

In Fig. 8, each graph models the behavior of data traffic between nodes during COMPS execution. In those graphs, each vertex pattern represents the volume of data read from that node by other workers, while each edge pattern represents the volume of data transferred between two nodes. In those scenarios, we only consider the traffic related to file access. To select such traffic in executions using the conventional system, we selected connections of the master COMPSs NIO connector (by default, port 43001) that sent the same amount of bytes of a fragment (64 MB). For executions using HDFS, we considered the connections where data was sent by the datanodes (by default, using port 50010).

The topology seen in the common file system case is different from both other graphs. Figure 8a shows that the master is the only node to read data from the file system, transferring it to the workers. Because of that, the master has the darkest pattern and workers have the lightest, while edges represent large upload volumes. In Fig. 8b-c, the master is responsible only for the orchestration of the tasks, no longer needing to transfer file contents. Because of that, significant data transfers are observed only between workers, but the vertex and edge patterns show that read load is distributed among all nodes, and that there are lower traffic volumes between any two nodes than on scenario (a). Finally, by increasing the locality of the data in scenario (c), data transfers drop drastically. There are still some traffic between nodes, but volumes are much lower, since they are mostly due only to the effect of reading logical blocks at the edges of the HDFS blocks (as explained in Fig. 2).

Table 3 shows the relative amount of data transferred through the network in those Grep executions. All columns show values relative to the size of the file (20 GB, in this case). In a COMPSs execution without a shared disk, on a conventional file system (case C in the Table 3), the input files are transferred from the master to all requesting workers. In other words, workers do not contribute in the transferring stage (0.0% of worker upload). This means that no fragments were located on the requesting computer, since COMPSs dictates that data be read sent by the main task. On the other hand, when using HDFS, data transfers are distributed over all workers (which also act as datanodes). When no locality-aware API was used and file blocks were not replicated (H1), most of the file data is still accessed over the network (89.3%), but each worker has to read only a fraction of that total (11.2%). Theoretically, the chance of a byte being accessed at the node node where it is stored is given by the ratio of favorable cases to total cases. Thus, in this scenario a given byte could be in only one of the eight machines, so it had a theoretical probability of 12.5% of being read *in-place*. That is close to the measured 10.7%. By increasing the replication factor to 3, there is a higher chance that a read will find a block locally at its own node, and the total traffic decreases (67.9%), as well as the amount of data uploaded by each node (8.6%). On the other hand, when we combine HDFS with the COMPSs storage PI, tasks tend to be assigned to workers located on the same nodes where data reside. Even when there is no replication (S1), only 11.1% of the data is accessed over the network, and on average each datanode has to serve only

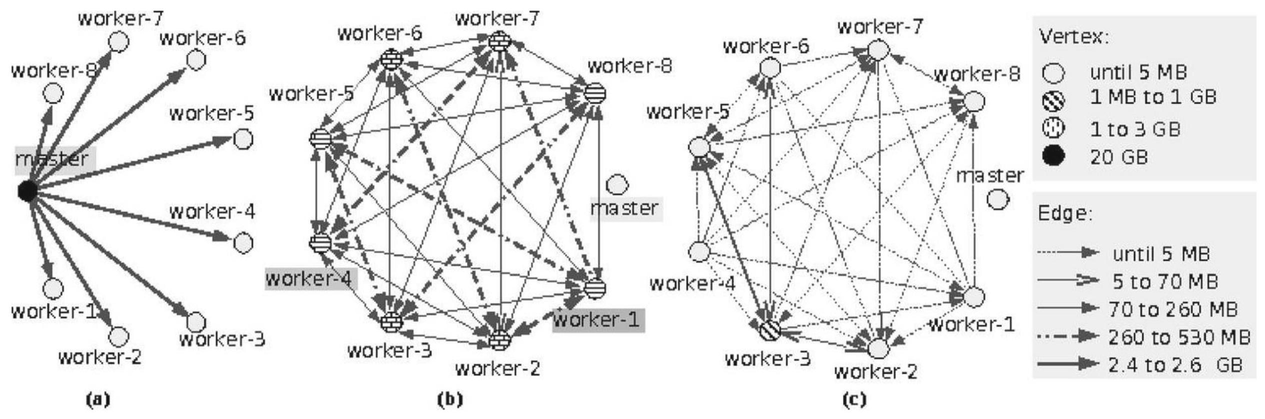


Fig. 8 Representation of network traffic patterns in COMPSs executions. **a)** conventional file system; **b)** HDFS API; **c)** HDFS+Storage APIs. The legend to the right indicates the aggregate volume of traffic read from a vertex or transferred through an edge. In the Conventional file system, all data is transferred between the master and the workers; by using HDFS, the traffic is distributed among workers, since all I/O is performed by directly by HDFS and data moves from the datanodes holding the data to the worker nodes where each block will be processed; when COMPSs Storage API is used, tasks are assigned preferentially to workers executing at the same nodes where blocks are stored, and network traffic is limited to a few cases where locality is not achievable (e.g., when reading records at the block edges)

Table 3 Network traffic of Grep on a cluster with eight workers

Case	% worker upload	% total traffic	% in place
C	0.0	100	0.00
H1	11.2	89.3	10.7
H3	8.6	67.9	32.1
S1	1.4	11.1	88.9
S3	0.16	1.7	98.3

Values are relative to the file size (20 GB). Except for the total traffic, values are the averages over all workers

1.4% of the file data to other nodes. When 3-way replication is considered, 98.3% of the data is read locally (using short-circuit techniques).

By this analysis, the use HDFS (even with the non-conventional use of replication factor 1) is preferable over the conventional file system. Although we only had 10.7% of the bytes executed in the node where they were stored, the transmission of the other data was distributed between all nodes of the cluster, avoiding a bottleneck at the master.

Another way to visualize the difference in performance is by analyzing traces created by COMPSs itself, as those shown in Fig. 9a and b. They represent the mapping of data transfers coordinated directly by COMPSs between cluster nodes during the execution of the Python versions of Wordcount with the HDFS API (scenario H1) and with a common file system (scenario C), for an 8 GB file. Red lines represent data transfers orchestrated by COMPSs during an execution.

In Fig. 9a, when the application uses the traditional file system, the master is responsible for the transfer of all data. We can see that as the set of lines fanning out from the master to all workers, starting soon after the execution starts. By the length of the lines we can see that part of the execution time is due to the wait for the file transfers to complete. All lines starting at the master are data transfers of parts of the input file to the workers. In this example there are more file fragments (64) than worker threads (4 in each worker, so 32 total), the master sends a first set of fragments soon after the application stars. When the tasks associated with those complete and return, the master goes on to send the 32 remaining fragments. Since the tasks do not complete at the same time, the messages for this second set are spread over a longer period of time (the block of messages around 105 s). Finally, the last set of messages, close to 163 s, are the messages related to the final reduction that adds all the partial counts

In Fig. 9b, the version using the HDFS API, we see only the data transfers orchestrated by COMPSs, since the HDFS access is not controlled directly by the COMPSs

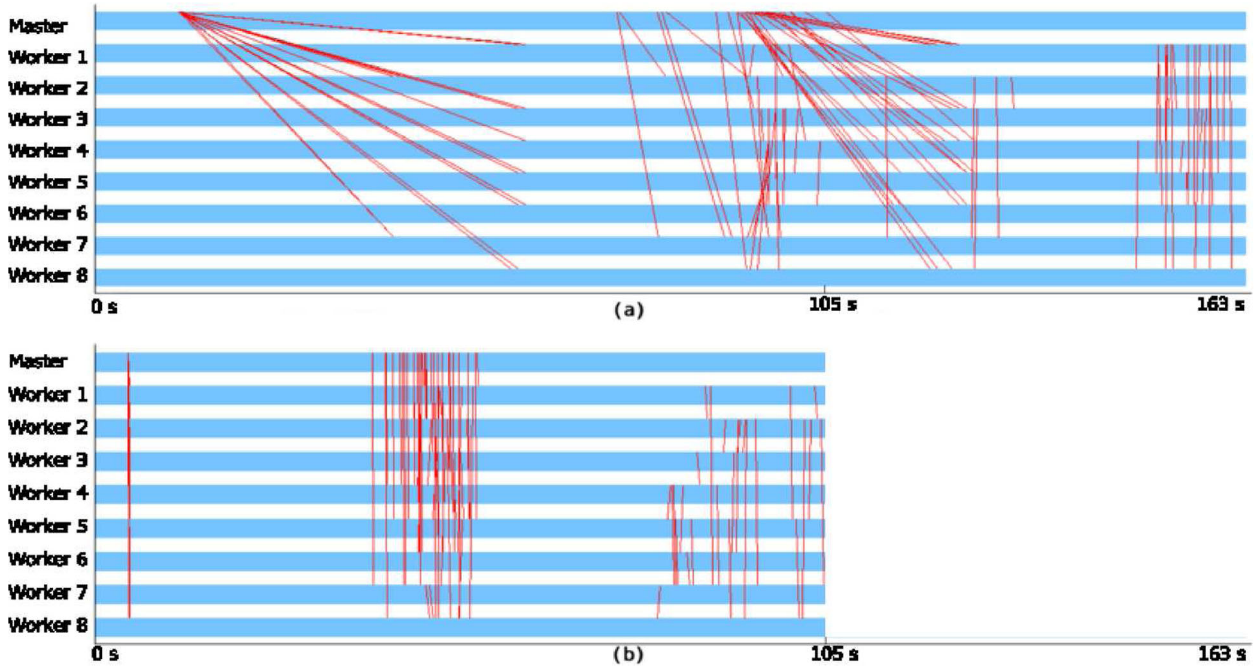


Fig. 9 Data transfer traces of COMPSs while running Wordcount. **a** Wordcount on a common file system (C): application ends around 163 s. **b** Wordcount using the HDFS API (H1): application ends in around 105 s. Blue bars represent the execution of the master and the 8 workers (identified to the left of the vertical axis). Red lines represent data transfers orchestrated by COMPSs during execution. They connect the bar of the two communicating nodes; horizontally, they indicate the duration of each transfer. Vertical lines represent very short data transfers, whose duration is not discernible at that time scale

runtime. Transfers are much shorter (almost vertical lines), since COMPSs only has to send the information about the list of fragments. Since we have twice the number of fragments than threads, we still have two moments when the master sends information about the fragments that need to be counted (the set of vertical lines at the beginning of execution, and the second set, around 50 s). HDFS is the one responsible for delivering the actual data to the multiple compute nodes. The last set of messages represents the reduction step.

6.4 Spark versus COMPSs

As previously mentioned in Section 2, a recent work [34] compared COMPSs performance in Java applications to Apache Spark, using a cluster architecture normally associated with HPC applications. Since our integration allows COMPSs to better interface with a cluster architecture more frequently found in Data Science scenarios, in this study we present a comparison of the two systems under those conditions,

Table 4 presents a performance comparison between COMPSs and Spark using the applications Grep, Wordcount, KMeans and KNN. The first two applications were implemented manually in COMPSs and executed in the different configurations mentioned previously (H3 and S3). The KMeans and KNN applications were created by Lemonade and executed in four scenarios, varying the HDFS interface (H3/S3) and the use of the optimization guidelines discussed in Section 5.3 (identified by the suffix *.opt* in the table). In our experiments, all Spark applications were implemented manually and used the same HDFS configuration as their COMPSs counterparts; Spark always takes data locality in consideration during execution. For each application, the *speedup* was computed in relation to the performance with Spark (which is represented as 1 in each case).

Table 4 Performance comparison: COMPSs versus Spark

Use case	Scenario	Time (s)	Speedup	95% CI
Grep	H3	59	2.74	(-∞, -101)
	S3	38	4.24	(-∞, -122)
	Spark	161	1	-
WC	H3	226	1.59	(-∞, -131)
	S3	210	1.71	(-∞, -148)
	Spark	358	1	-
KMeans	H3	905	0.48	(472, ∞)
	H3.opt	571	0.77	(138, ∞)
	S3.opt	438	1.00	(-6, 5)[†]
	Spark	438	1	-
KNN	H3	1426	0.52	(691, ∞)
	H3.opt	1076	0.69	(340, ∞)
	S3.opt	711	1.05	(-∞, -23)
	Spark	746	1	-

[†] Difference not statistically significant

The column 95% Confidence Interval of Table 4 (95% CI) represents the result of the paired Z test between Spark and COMPS scenarios; all ranges are one-sided, except for *KMeans (S3.opt)*, which is two-sided because there was no significant difference between those systems. According to the table, we can see that the first two applications had better performance with COMPSs than with Spark, regardless of the API used. The result in that table, with the analysis presented earlier for Fig. 6, suggests that HDFS contributes to the increased performance of COMPSs, making it competitive with the Spark solution. In the Grep application, for example, COMPSs obtained a speedup of 4.24, being at least 122 s faster than Spark.

Spark has well-known mechanisms for code optimization and data locality since its first versions, while COMPSs has the Storage API to exploit data locality. However, the user is still responsible for tuning the code and, as shown in Table 4, a good implementation can have an impact on COMPSs executions. For instance, a well-implemented, optimized KNN workflow (*H3 - opt*), is 1.33 faster than a naive implementation (*H3*). We also showed that Lemonade+COMPSs is able to match the Spark performance in complex scenarios like KMeans or KNN, characterized by several stages of tasks, even with loops. In fact, in the KMeans scenario, when Lemonade generates a better implementation (i.e., with code optimization guidelines and using the locality-aware version of the HDFS API), the two-side 95% confidence interval indicates that there was no significant performance difference. In the KNN case, although we obtained a speedup of 1.05, the one-side interval of (-∞, -23) suggests that this COMPSs superiority is significant, although not very large. Those results suggest that COMPSs is a powerful framework and that the current version of Lemonade, using the initial optimization guidelines and the HDFS API with the locality-aware COMPSs storage API is able to generate good Python implementations in COMPSs with performances comparable to Spark in Big Data scenarios.

7 Conclusion

Advances in the HPC and big data areas have led to the development of techniques that are being used in both of them. In this work we proposed and evaluated new extensions to the COMPSs environment, originally used in HPC applications, to increase its performance and facilitate its application in big data scenarios. These open-source extensions allow its integration with a distributed file system (HDFS) and a visual tool for Data Analytics (Lemonade).

We have shown that in a typical big data scenario, with a Horizontal scaling architecture, conventional networks and no shared disk solution, the use of HDFS is indispensable for better performance. Also, the use of HDFS in COMPSs applications is recommended, not only by the already known HDFS advantages, but also because it provides a data abstraction, the division of data in blocks, that helps to express in COMPSs algorithms from Machine Learning and Data Mining that deal with large volumes of data.

Lemonade has aspects, such as a friendly visual user interface for creating and executing flows using the drag and drop elements, that justify its use. We have shown that Lemonade is able to generate efficient cods in COMPSs that achieve performances comparable to Spark. In addition, beginner or advanced COMPSs programmers can use the algorithms implemented for Lemonade as an external library for their applications, even if they do not want to use Lemonade itself.

Our ongoing work includes experimental tests to evaluate the HDFS extension with bigger data sets and other real scenarios. We also plan to support more operations and algorithms in Lemonade and to improve our Lemonade optimization guidelines. To accomplish that we will examine the possibility to generate more flexible Lemonade code to, for example, when possible, re-define the number of fragments in runtime based on the size of the data being produced. We expect that the generated code in that case will be better fitted to what is being processed, optimizing the number of tasks created, which may also be beneficial to COMPSs in general.

Abbreviations
API: Application programming interface; ETL: Extract transform load; HDFS: Hadoop distributed file system; HPC: High-performance computing; KNN: K nearest neighbors; ML: Machine learning

Acknowledgements
Not applicable.

Authors' contributions
All authors read and approved the final manuscript.

Funding
This work was partially funded by Fapemig, CAPES, CNPq, MCT/CNPq-InWeb, FAPEMIG-PRONEX-MASWeb (APQ-01400-14), and by the collaboration between Brazilian MCT/RNP and the European Union Horizon 2020 research and innovation programme under grants 690116 (EUBra-BIGSEA) and 777154 (EUBra Atmosphere).

Availability of data and materials
As mentioned in the text, the implemented systems are open source, available through GitHub. Readers can contact the corresponding author to request the dataset used in this work.

Competing interests
The authors declare that they have no competing interests.

Author details
[1]Departamento de Ciência da Computação, Universidade Federal de Minas Gerais (UFMG), 31270-901 Belo Horizonte, Minas Gerais, Brazil. [2]Barcelona Supercomputing Center (BSC-CNS), Barcelona, Spain. [3]Artificial Intelligence Research Institute (IIIA), Spanish Council for Scientific Research (CSIC), Barcelona, Spain.

Received: 15 February 2019 Accepted: 5 September 2019
Published online: 16 October 2019

References
1. Kamburugamuve S, et al. Twister2: Design of a big data toolkit. Concurr Comput: Pract Experience. 2019;31(14). https://doi.org/10.1002/cpe.5189.
2. Fox G, et al. Big data, simulations and HPC convergence. In: Big Data Benchmarking: 6th International Workshop, WBDB 2015, Toronto, ON, Canada, June 16-17, 2015 and 7th International Workshop, WBDB 2015, New Delhi, India, December 14-15, 2015, Revised Selected Papers. Cham, Switzerland: Springer; 2016. p. 3–17. https://doi.org/10.1007/978-3-319-49748-8_1.
3. Tejedor E, et al. PyCOMPSs: Parallel computational workflows in Python. Int High Perform Comput Appl. 2017;31(1):66–82. https://doi.org/10.1177/1094342015594678.
4. Asch M, et al. Big data and extreme-scale computing: Pathways to convergence-toward a shaping strategy for a future software and data ecosystem for scientific inquiry. Int J High Perform Comput Appl. 2018;32(4):435–79. https://doi.org/10.1177/1094342018778123.
5. Lezzi D, et al. Enabling e-Science applications on the cloud with COMPSs. In: Parallel Processing Workshops at European Conference on Parallel Processing (Euro-Par 2011). Berlin: Springer; 2011. p. 25–34. https://doi.org/10.1007/978-3-642-29737-3_4.
6. Lordan F, Ejarque J, Sirvent R, Badia RM. Energy-aware programming model for distributed infrastructures. In: 24th Euromicro International Conference on Parallel, Distributed, and Network-Based Processing (PDP 2016). Washington: IEEE Computer Society; 2016. p. 413–7. https://doi.org/10.1109/pdp.2016.39.
7. Zaharia M, et al. Resilient distributed datasets: A fault-tolerant abstraction for in-memory cluster computing. In: Proceedings of the 9th USENIX Symposium on Networked Systems Design and Implementation (NSDI '12). Berkeley: USENIX Association; 2012. p. 15–28. https://dl.acm.org/citation.cfm?id=2228301.
8. Santos W, et al. Lemonade: A scalable and efficient Spark-based platform for Data Analytics. In: 17th IEEE/ACM International Symposium on Cluster, Cloud and Grid Computing (CCGRID). Piscataway: IEEE Press; 2017. p. 745–8. https://doi.org/10.1109/CCGRID.2017.142.
9. Marozzo F, et al. Enabling cloud interoperability with COMPSs. In: Parallel Processing Workshops at European Conference on Parallel Processing (Euro-Par 2012). Berlin: Springer; 2012. p. 16–27. https://doi.org/10.1007/978-3-642-32820-6_4.
10. Ramon-Cortes C, et al. Transparent orchestration of task-based parallel applications in containers platforms. 2018;16(1):137–60. https://doi.org/10.1007/s10723-017-9425-z.
11. Apache Cassandra. http://cassandra.apache.org/. Accessed 4 July 2019.
12. Shepler S, Eisler M, Noveck D. Network file system (NFS) version 4 minor version 1 protocol. RFC. 2010;5661:1–617. https://doi.org/10.17487/RFC5661.
13. Li H. Alluxio: A virtual distributed file system. EECS Department, University of California, Berkeley, USA. 2018. http://www2.eecs.berkeley.edu/Pubs/TechRpts/2018/EECS-2018-29.html.
14. Amazon Simple Storage Service (S3). https://aws.amazon.com/s3/. Accessed 4 July 2019.
15. Microsoft Azure Storage. https://azure.microsoft.com/services/storage/. Accessed 4 July 2019.
16. Schwan P. Lustre: Building a file system for 1000-node clusters. In: Proceedings of the Linux Symposium. Ottawa: Linux symposium; 2003. p. 380–6. https://www.kernel.org/doc/ols/2003/ols2003-pages-380-386.pdf.
17. OpenStack Storage (Swift). https://docs.openstack.org/swift/. Accessed 4 July 2019.
18. Weil SA, et al. Ceph: A scalable, high-performance distributed file system. In: Proceedings of the 7th Symposium on Operating Systems Design and Implementation (OSDI '06). Berkeley: USENIX Association; 2006. p. 307–20. http://dl.acm.org/citation.cfm?id=1298455.1298485.
19. Andersen DG, et al. FAWN: A fast array of wimpy nodes. In: Proceedings of the 22nd ACM SIGOPS Symposium on Operating Systems Principles

(SOSP '09). New York: ACM; 2009. p. 1–14. https://doi.org/10.1145/1629575.1629577.
20. DeCandia G, et al. Dynamo: Amazon's highly available key-value store. In: Proceedings of the 21st ACM SIGOPS Symposium on Operating Systems Principles (SOSP '07). New York: ACM; 2007. p. 205–20. https://doi.org/10.1145/1294261.1294281.
21. Memcached: A distributed memory object caching system. http://memcached.org/.. Accessed 4 July 2019.
22. Apache HBase. http://hbase.apache.org/.. Accessed 4 July 2019.
23. Palankar MR, et al. Amazon S3 for science grids: A viable solution? In: Proceedings of the 2008 International Workshop on Data-aware Distributed Computing (DADC '08). New York: ACM; 2008. p. 55–64. https://doi.org/10.1145/1383519.1383526.
24. Wickramasinghe P, et al. Twister2:TSet high-performance iterative dataflow. In: International Conference on High Performance Big Data and Intelligent Systems (HPBD&IS 2019). Piscataway: IEEE Press; 2019. p. 55–60. https://doi.org/10.1109/HPBDIS.2019.8735495.
25. Goodstadt L. Ruffus: a lightweight Python library for computational pipelines. Bioinformatics. 2010;26(21):2778–9. https://doi.org/10.1093/bioinformatics/btq524.
26. Gafni E, et al. COSMOS: Python library for massively parallel workflows. Bioinformatics. 2014;30(20):2956–8. https://doi.org/10.1093/bioinformatics/btu385.
27. Mierswa I, et al. YALE: Rapid prototyping for complex data mining tasks. In: Proceedings of the 12th ACM SIGKDD International Conference on Knowledge Discovery and Data Mining. New York: ACM; 2006. p. 935–40. https://doi.org/10.1145/1150402.1150531.
28. Demšar J, et al. Orange: Data mining toolbox in Python. J Mach Learn Res. 2013;14(1):2349–53.
29. Berthold MR, et al. KNIME - the konstanz information miner: version 2.0 and beyond. ACM SIGKDD Explor Newsl. 2009;11(1):26–31. https://doi.org/10.1145/1656274.1656280.
30. Dean J, Ghemawat S. Mapreduce: Simplified data processing on large clusters. In: OSDI'04: Sixth Symposium on Operating System Design and Implementation. San Francisco: USENIX Association; 2004. p. 137–50.
31. Kranjc J, et al. ClowdFlows: A cloud based scientific workflow platform. In: Machine Learning and Knowledge Discovery in Databases: European Conference (ECML PKDD 2012). Berlin: Springer; 2012. p. 816–9. https://doi.org/10.1007/978-3-642-33486-3_5.
32. Podpečan V, Zemenova M, Lavrač N. Orange4WS environment for service-oriented data mining. Comput J. 2012;55(1):82–98. https://doi.org/10.1093/comjnl/bxr077.
33. Microsoft Azure Machine Learning. https://azure.microsoft.com/services/machine-learning-studio/.. Accessed 4 July 2019.
34. Conejero J, et al. Task-based programming in COMPSs to converge from HPC to big data. Int J Perform Comput Appl. 2018;32(1):45–60. https://doi.org/10.1177/1094342017701278.
35. White T. Hadoop: The Definitive Guide, 4th. Sebastopol: O'Reilly Media, Inc.; 2015.
36. Gonzales SD. PyWebHDFS: a Python wrapper for the Hadoop WebHDFS REST API. 2016. https://pypi.python.org/pypi/pywebhdfs/.. Accessed 4 July 2019.
37. Luckow A. WebHDFS: HDFS Python client based on WebHDFS REST API. 2014. https://pypi.org/project/WebHDFS/.. Accessed 4 July 2019.
38. Kalika M. Python WebHDFS. 2019. https://github.com/mk23/webhdfs.. Accessed 4 July 2019.
39. Rosen J. PySpark Internals. 2016. https://cwiki.apache.org/confluence/display/SPARK/PySpark+Internals/.. Accessed 4 July 2019.
40. Leo S, Zanetti G. Pydoop: a Python MapReduce and HDFS API for Hadoop. In: Proceedings of the 19th ACM International Symposium on High Performance Distributed Computing. New York: ACM; 2010. p. 819–25. https://doi.org/10.1145/1851476.1851594.
41. Apache Arrow Developers. Pyarrow: Python library for Apache Arrow. 2016. https://pypi.org/project/pyarrow/.. Accessed 4 July 2019.
42. Chang L, et al. HAWQ: A massively parallel processing SQL engine in Hadoop. In: Proceedings of the 2014 ACM SIGMOD International Conference on Management of Data (SIGMOD '14). New York: ACM; 2014. p. 1223–34. https://doi.org/10.1145/2588555.2595636.
43. McKinney W. Pandas: a foundational Python library for data analysis and statistics. In: Workshop on Python for High Performance and Scientific Computing Collocated with the 24rd International Conference for High Performance Computing, Networking, Storage and Analysis (SC '11). New York: ACM; 2011.
44. Jain R. The Art of Computer Systems Performance Analysis: Techniques for Experimental Design, Measurement, Simulation, and Modeling. Wiley Computer Publishing. New York: Wiley; 1991.

Publisher's Note