DDF Library: enabling functional programming in a task-based model

Lucas M. Ponce[a,*], Daniele Lezzi[b], Rosa M. Badia[b,c], Dorgival Guedes[a]

[a] *Universidade Federal de Minas Gerais (UFMG), Belo Horizonte, Brazil*
[b] *Barcelona Supercomputing Center - Centro Nacional de Supercomputación (BSC-CNS), Barcelona, Spain*
[c] *Spanish National Research Council (CSIC), Barcelona, Spain*

Abstract

In recent years, the areas of High-Performance Computing (HPC) and massive data processing (also know as Big Data) have been in a convergence course, since they tend to be deployed on similar hardware. HPC systems have historically performed well in regular, matrix-based computations; on the other hand, Big Data problems have often excelled in fine-grained, data parallel workloads. While HPC programming is mostly task-based, like COMPSs, popular Big Data environments, like Spark, adopt the functional programming paradigm. A careful analysis shows that there are pros and cons to both approaches, and integrating them may yield interesting results. With that reasoning in mind, we have developed DDF, an API and library for COMPSs that allows developers to use Big Data techniques while using that HPC environment. DDF has a functional-based interface, similar to many Data Science tools, that allows us to use dynamic evaluation to adapt the task execution in run time. It brings some of the qualities of Big Data programming, making it easier for application domain experts to write Data Analysis jobs. In this article we discuss the API and evaluate the impact of the techniques used in its implementation that allow a more efficient COMPSs execution. In addition, we present a performance comparison with Spark in several application patterns. The results show that each technique significantly impacts the performance, allowing COMPSs to outperform Spark in many use cases.

Keywords: COMPSs, Big Data, Performance Evaluation, Data-Flow Programming

1. Introduction

Traditionally, Big Data and HPC systems are quite different. Big data is usually related to a high-level programming models, based on MapReduce [1],

*Corresponding author
Email address: lucasmsp@dcc.ufmg.br (Lucas M. Ponce)

resources managers like YARN and file systems like HDFS [2], using a local shared-nothing architecture. However, in HPC systems prevail resources managers like Slurm [3], file systems like Lustre [4], in a supercomputer architecture based on remote shared parallel storage. For a long time, each of these technologies has been applied only to its specific niches. Lately, the convergence between HPC and Big Data has become an important research area, driven in part by the need to incorporate high-level libraries, platforms, and algorithms for machine learning and graph processing, and in part by the idea of using Big Data's fine-grained data awareness to increase the productivity of HPC systems [5, 6]. Several proposals of higher-level abstractions have emerged to address the requirements of these two areas in computer systems [7, 8]. Recent frameworks, like COMPSs [7], Twister2 [8], Spark [9] and Flink [10], share a common dataflow programming model, but each one is still focused on its respective area.

Dataflow is a special case of task-based models where an application can be represented as a directed acyclic graph (DAG), with nodes representing computational steps and edges indicating communication between nodes. The computation at a node is activated when its inputs (events, task data) become available. A well-designed dataflow framework hides low-level operational details, such as communication, concurrency control, and disk I/O, from the users developing parallel applications, allowing them to focus on the application itself.

While sharing that common model, each framework has its own abstraction and run time system, which are generally related to its original environment. Traditionally, HPC environments provide an interface through User-Defined Functions, which gives freedom to write their applications by defining its tasks. For instance, in COMPSs, an MPI-based framework commonly used in HPC scenarios, applications 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 and what are its inputs and outputs. Such frameworks are commonly used in scientific algorithms such as matrix computations. Despite the good performance in those scenarios, it is often hard to implement optimized applications that handle irregular data and complex data flows, such as those commonly found in machine learning and data mining areas.

The process of transmitting large volumes of input data to tasks has a high cost in many HPC systems, specially when those data are the output of a previous task, as it is the case in COMPSs, because it involves data serialization and de-serialization steps. Because of that, a common practice adopted by advanced programmers is to minimize the number of different tasks by combining the code of multiple functions in a single task. This is a challenge when black-box libraries of parallel algorithms are used, because, depending on the flow of operations, it might be necessary to merge the code of different functions in order to obtain better performance, but that code would not be available.

On the other hand, recent Big Data environments have adopted functional languages [9, 10] as a form to express their data abstractions and flows. Those frameworks implement a set of common operations and basic algorithms to facilitate the development of applications by experts in the application domain.

However, some research shows that, depending on the application (*e.g.*, matrix computations), those frameworks achieve good scalability, but poor performance when compared with an MPI implementation [11, 12].

In that context, our work discusses our experience in bringing together the programming model of COMPSs and the functional programming abstractions usually found in frameworks like Spark. Our contributions to the convergence path between HPC and Big Data frameworks are: (*i*) a discussion of different implementation techniques used in recent dataflow models to build more optimized systems and their evaluation in COMPSs; (*ii*) an API, in the form of the DDF Library, which materializes our vision of a big data analytic tool that runs on top of an HPC framework to execute applications efficiently; and (*iii*) a performance comparison of COMPSs and Spark applications using Python.

The goal of DDF is to provide users with performance comparable to HPC systems while exposing a well-known user-friendly dataflow abstraction for application development. We chose to work on COMPSs, extending it with DDF, because it is a good framework for Data Scientists that want to create and execute Big Data applications: it has been gaining popularity, it supports high-level languages like Python, it has a generic data model, which allows it to be extended more easily, and it has good performance [11].

To describe our work, the remainder of the paper is structured as follows: Section 2 presents some related work; Section 3 introduces the COMPSs framework and Section 4 presents some optimization techniques; Section 5 presents our API, which provides a new data abstraction and interface to COMPSs users. The validation of our solution is discussed in Section 6, and Section 7 presents our conclusions and discusses future work.

2. Related work

While dataflow is a prevalent model in many parallel and distributed programming frameworks [8], functional programming is slowly becoming a commom interface. In addition to Big Data frameworks like Spark, Flink and Swift [13], functional interfaces are also being frequently used in other Data Sciences programming tools (*e.g.*, Scikit-Learn [14] and Pandas [15] use it to express their dataflow models).

In the Big Data field, Spark is probably the framework that most contributed to the popularization of the functional interface. It was originally built on top of the Resilient Distributed Dataset abstraction (RDD), a read-only multiset of objects partitioned across multiple nodes that holds provenance information (lineage). More recently, since Version 2.4, Spark added the DataFrame, an abstraction equivalent to a table in a relational database, built on top of the RDD. By working with structured data, the DataFrame allowed Spark to gain performance using an optimized execution engine [16]. Besides a set of RDD/DataFrame operators, Spark offers other tools and libraries for machine learning, graph analytics and stream processing, among others. In particular, it provides the MLlib [17] and ML machine learning libraries, but on top of RDD and DataFrames, respectively.

In past years, many proposed research tried to increase Spark's performance in order to make it competitive with HPC frameworks. One of them is Spark-DIY [18], where authors created a prototype framework with the integration of an MPI layer into Spark. The prototype is based on overloaded Spark RDD operators with MPI-based implementations (DIY). Although the authors showed a performance gain by using DIY, Spark's usability is affected: users need to provide their own MPI code to replace a given operator.

In the effort to support functional language constructs, existing frameworks have been extended to make them more attractive to users who are already familiar with that interface, such as the TSet abstraction for Twister2 framework [8], DDS [19] and DisLib [20] for COMPSs. In the case of Twister2, the goal of TSet was to provide users with performance of an HPC framework while exposing a user-friendly dataflow abstraction, similar to Spark's RDD, in Java. Although the operators provided are limited, the authors showed that Twister2 outperforms Spark in algorithms like KMeans and SVM, that can be written using those operators. DDS and DisLib are the first official efforts of the COMPSs team to enable large-scale data analytics on HPC infrastructures by providing an abstraction similar to the RDD and a library like Scikit-Learn, respectively. Using the DDS interface, applications can be written using operators like `load, map, filter, and reduce`, also similar to Spark's RDD, and using Dislib, users can execute machine learning algorithms. Although both projects are in Python, they are not integrated: each project relies on its own data abstraction, so users might need to convert their data representations to use both frameworks.

New frameworks are also being proposed on this same premise. A project that is gaining a lot of popularity at the moment is Dask [21], a framework that provides data abstractions for n-dimensional arrays and DataFrames that can be operated in parallel in a transparent manner. The Dask project includes Dask-ML [22], a library that provides many machine learning algorithms through an estimator-based interface. Dask is mainly used to scale up on all cores of a single machine. One of its advantages is the integration with Scikit-Learn's algorithms through the Joblib backend [23] which allows for scaling out CPU-bound workloads; workloads with datasets that fit in RAM, but have many individual operations, can be run in parallel. However, to scale out RAM-bound workloads (larger-than-memory datasets) the solutions are more limited. For instance, in that case Dask-ML recommends some approaches: (*i*) use the `ParallelPostFit` method to distribute the execution of an already trained mode to the various fragments of data in Dask, which does not parallelize the training step; or (*ii*) use Incremental estimators which, although it allows to fit a model on large volumes of data by incremental training from each chunk of the data, leads to training that is not parallelized. For this reason, Spark remains a more popular option when it comes to Big Data scenarios.

The DDF Library we present here resembles TSet, DDS, Dislib, and Dask by providing a new auxiliary data abstraction, the DDF, for COMPSs to handle Big Data. Like Dask, we adopted a DataFrame abstraction, an increasingly popular structure in Data Science [24, 25]. However, we implemented a large

set of operations and algorithms, many of them not available in DDS nor Dislib. As in Spark, the central idea of the supported algorithms and operations is the parallelization of operations on data fragments. We work on Big Data scenarios, where we assume that we cannot fit all data in a single memory node. In addition, all available algorithms are integrated in the same interface and use a context manager to submit dynamical tasks.

This paper extends previous work [26] by providing a detailed description of the library, discussing implementation techniques not presented before, as well by adding as some new analysis of the impact of those techniques, and new performance evaluation results.

3. The COMPSs framework

COMPSs is a programming framework whose main objective is to simplify 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 code annotations 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, among other information, whether a parameter is an input or output. In the case of Python (PyCOMPS), tasks are identified with an annotation in the form of a decorator started with `@task` on top of a method. With that information, COMPSs generates a task graph at execution time where each node denotes a task, and edges between them represent data dependencies. The task graph expresses the inherent parallelism of the application at task level, which is used by the runtime.

The COMPSs runtime architecture is based on a main component, the master, which 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.

Regarding the programming model, to port an application to COMPSs, besides requiring the identification of the functions as tasks, may require structural changes to the code in order to improve application efficiency and to 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. A much more efficient approach in COMPSs, which exploits a higher level of parallelism, if there are no dependencies among file data elements, 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. COMPSs is able to transfer, transparently, files that are used as input parameter for a task;

it also supports shared disks or HDFS (by using a connector [27]) to speed up the read step.

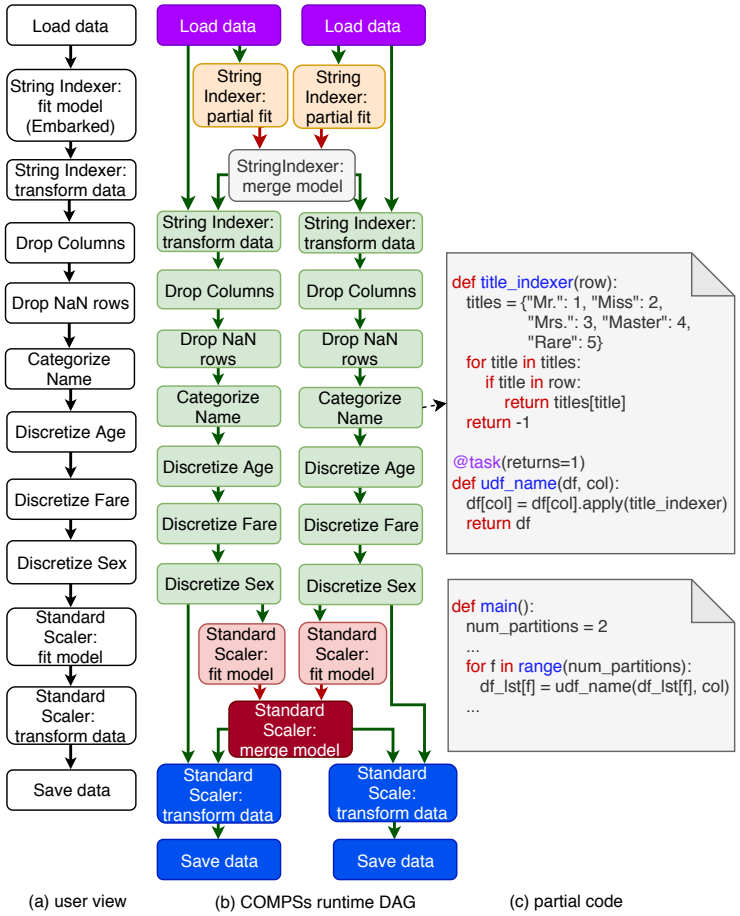

Figure 1: Preprocessing Titanic's data set in PyCOMPSs.

As an example, Fig. 1(a) shows the visual workflow of an application that performs a preprocessing step to predict survival on the Titanic Disaster[1]. The idea for this implementation is to break the input in two partitions and to process them concurrently, when possible; the dependency graph, similar to the one produced during execution, is shown in Fig. 1(b). Some operations are embarrassingly parallel and do not need more than one stage for each partition, like `Categorize Name`. Others, however, like `String Indexer`, need more than one stage, since they have some actions that depend on the combination of

[1]based on: `https://bit.ly/2MyBOpa`, which uses data from `https://www.kaggle.com/c/titanic`

partial results obtained from each partition. To illustrate the use of PyCOMPSs, Fig. 1(c) shows part of the code, including the `Categorize Name` step, which converts passengers names to an index based on its title ("Mr." will be mapped to 1, "Miss" to 2, etc.). That can be done using function `udf_name` on each partition. The `@task` annotation indicates that the function is a COMPSs task which returns one output. The result `df` will be saved and sent to the next task transparently. COMPSs provides several tags for better specification, for example, the `FILE_OUT` tag can be used to inform that a given result is a file.

4. Optimization techniques

In distributed systems, the transmission of volumes of data between workers is one of the main factors that affect performance. This process often involves another costly step, the serialization of data into formats that can be transmitted and later interpreted (de-serialized) by the receiver. General purpose frameworks, such as COMPSs and Spark, focus on minimizing the amount of serialized data, or reducing the number of data transfers. In the sections that follow, we describe the optimization techniques that can be used for those frameworks.

4.1. Serialization

When dealing with Big Data frameworks, data type compatibility, read and write speed, and compression capability are the major aspects to consider when choosing a serialization method. For generic environments, the compatibility with different types of data is a crucial point. For example, due to the characteristic of RDDs in Spark, which can contain any type of data, the standard solution is the Java serialization. Although Spark supports Kyro, another serialization method that is faster and more compact than the default, it is not compatible with all the possible data types [28].

On the other hand, when we restrict the scope, for instance, by working with structured data such as DataFrames, new opportunities for serialization techniques become available. Nowadays, columnar file formats are well-known solutions to store structured data in a column-oriented way. For instance, when Spark works with DataFrames, internally it uses Parquet, a format inspired by the Google Dremel framework [29], while Apache Hive, a big data framework for data warehouses, uses the ORC format [30]. Organizing data by columns, unlike traditional row-oriented formats, allows chunks of data of the same type to be stored sequentially. Thus, the encoding and compression algorithms can take advantage of the data type knowledge and homogeneity to achieve better efficiency both in terms of speed and file size [31]. Besides that, many columnar formats allow efficient scans when only a subset of the columns is considered. Because COMPSs does not provide a native data abstraction for structured data, it does not support such approaches.

4.2. Grouping tasks

Dataflow frameworks need handle data and control transfers to switch from one task to the next when processing a flow. In COMPSs, for instance, at the end of a task its output is always serialized and saved to disk until some other task requires it. In Spark, results are serialized and saved in memory; only when that is not possible the result is written to disk. Whatever the procedure adopted, the context change between tasks always causes overhead. Building an efficient runtime depends on minimizing that overhead by reducing the number of different tasks. To be able to do that, we must characterize the way tasks depend on each other. Like Spark, we define dependencies as *narrow*, when each instance of a task depends on at most one instance of each of its parents (green arrows in Fig. 1(b)); or as *wide*, when that is not the case (red arrows in Fig. 1(b)). To filter or drop rows, and to replace values, are examples of *narrow* dependencies; to sort data, to perform aggregations/joins and to find duplicate elements are *wide* dependencies. In the Titanic application, the `Categorize Name` task produces one output item directly for each input, so it is has a *narrow* dependency. However, the `String Indexer` has a *wide* dependency, because it needs to create a global result based on all its partial inputs.

Experienced programmers group sets of tasks that have *narrow* dependencies in a single set: since those tasks do not need data from other partitions, that set can be executed as a single pipeline. For example, in the Titanic application DAG, tasks of the same color can be grouped; in that example, the `Standard Scaler: transform data` and `Save data` tasks could be performed together. In case of a bifurcation (*i.e.*, when the output of a task is used in two distinct operation flows), that grouping would have to be interrupted at that level and a serialization would be necessary for that step. Spark adopted that technique internally, by grouping sets of *narrow* tasks into a unit called a *Stage*. On the other hand, in COMPSs, the implementation of each task is the responsibility of the programmer, so it does not know how to classify tasks *a priori*.

4.3. Lazy evaluation

Frameworks generally create and analyze a DAG of tasks based on the code provided to decide when a set of tasks can be grouped. However, in interactive environments, which are often used for Data Science exploratory tasks, the complete code is not always available beforehand. Thus, it is often not possible to make decisions based on the code available so far. Lazy evaluation is a technique used in frameworks based on functional language to delay the execution of a task until the user actually needs its result. In general, operations submitted by a user are added to a queue until a certain condition is met. In Spark, if a data transformation operation has *narrow* dependencies on its parents, it is added to a queue with them, creating a *Stage*. When processing is required, like when an action (operation that returns information to the user) is submitted, the enqueued tasks are executed. That technique allows frameworks to analyze and optimize the flow of operations.

4.4. Repartitioning to minimize data shuffle

Wide dependency tasks generally need to collect data generated in another worker. For example, consider an inner join operation of two large tables (*T1* and *T2*) in a scenario where each table is divided into four fragments. A naive strategy to join them would be to compare each fragment of *T1* with each fragment of *T2*; the merged result would correspond to an exact solution. However, it would be necessary to create 16 partial inner join tasks to create that result. The naive inner join is an expensive operation because, in addition to the computational cost of having to do 16 inner joins, it incurs in network transfer, serialization and de-serialization costs.

One smarter approach to minimize the cost of *wide* dependencies is to reorganize (re-partition) the data as part of the process, to reduce communication. The two most common partitioning modes are hash and range partitioning. In the first one, given a set of keys (which will be used in the inner join), the new partition index of each element is defined by its hash code. In the second, each partition must establish a range condition using key values. The idea is similar to MapReduce's shuffle step [1], where it re-organizes its data before the reduce step.

Fig. 2 exemplifies a generic type of efficient partitioning. Consider, for example, a repartitioning of an initial set of two fragments (*p1* and *p2*). Each fragment will be sub-divided into two new fragments based on the color of each element. Once this step is completed, sub-fragments with the same index are merged (*i.e.*, all *sub-p1* will be merged). In this example, the number of final fragments is equal to the initial number, but it could be different, if desired. It is important to note that only sub-fragments are transferred over the network in the second step (sub-fragment junction). The transfer of each sub-fragment occurs only once and each of them tends to be smaller than the original fragment.

This idea can also be extended by reducing the data before re-partitioning. For instance, the operation of removing duplicated rows based on their keys requires shuffling data between partitions. However, instead of re-partitioning the raw data, a more beneficial approach might remove the duplicated keys in each partition, reducing the amount of data to be re-partitioned. Another case that can be optimized is when a shuffle occurs before an operation that reduces data. For instance, a flow that contains a filter following a sort operation. In this example, a better approach would be to filter the data before the sort operation. Recent frameworks, like Spark, use their functional-based interface to hide all the complexity of optimizing parallel code by analyzing the flow of the operations and re-organizing its tasks.

4.5. Exploring data locality

Besides the mentioned techniques, another way to decrease the transfers between nodes is to explore data locality by scheduling tasks on nodes that already possess the input data. Recent frameworks, such as COMPSs and Spark,

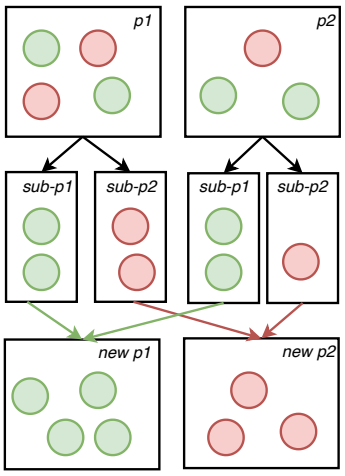

Figure 2: Exemplification of an partitioning based on a condition.

implement different schedulers to explore data locality and other policies transparently to users. In addition, distributed storage systems (*e.g.*, HDFS, Cassandra, Hive, among others) that are supported in many frameworks like Spark (natively) or COMPSs (through an API [27]) can help the schedulers by increasing the possibilities of improving data locality when reading files. Frameworks that use a conventional file system typically adopt as a rule that input files will be located on the master computer and will be transferred over the network when required by a task. When using HDFS, for instance, data is distributed over nodes and replicated to increase data availability; that information can be used by schedulers to direct execution to the best data providers in each case.

4.6. Integrating pre-compiled code in applications

Python is an easy-to-use language that has been gaining momentum in recent years in scientific computing, sometimes replacing traditional tools as Matlab [7]. However, there are well-known factors (*e.g.*, the absence of strong typing), that can significantly limit its performance. As a solution, many libraries such as NumPy [32], Pandas and Scikit-Learn, provide a set of Python high-performance operations using pre-compiled functions based on C/C++. When implementing an application, it is expected that users use the maximum amount of pre-implemented functions, also called vectorized functions, in contrast to pure Python code, to speedup their applications. Spark adopts a similar idea: its algorithms and operations available in PySpark (Spark using Python) are executed in Scala through a Java Virtual Machine (JVM) connector in the Spark runtime.

5. DDF

Based on our experience developing Data Science applications and the approaches discussed in Section 4, we developed DDF (Distributed DataFrames), a high-level data abstraction for COMPSs applications with a functional language interface, its extensions to the COMPSs runtime and an initial library of algorithms for Python. DDF currently includes approximately 40 Extract-Transform-Load (ETL) operations (*e.g.*, data set union, data load, drop columns and rows, joins, sort) and more than 30 machine learning algorithms (including scalers, classifiers, regressions and clustering algorithms), all publicly available[2].

DDF is based on the abstraction of the DataFrame, similar to Spark's and Panda's DataFrame, where data is distributed over nodes. Similar to those tools, DDF expresses operations by using operators that hide all code related to those tasks and the optimization policies. It runs on top of Pandas, a library providing high-performance, easy-to-use data structures and data analysis tools for Python. Figure 3 shows the internal structure of the DDF class. It abstracts its data as a list of n DataFrames that represents the data fragmented in n parts. Using Panda's abstraction allows us to use a wide set of well-implemented and documented functions. However, there is no fixed relationship between functions provided by Pandas and by DDF, because working in a distributed environment may require additional operations. For instance, to sort data in DDF, internally the data is re-partitioned as mentioned in Section 4 before sorting. Also, some algorithms available in DDF use NumPy functions to speed up Python execution by using well-implemented C/C++ functions.

Fig. 4 shows the code of the Titanic application, previously mentioned in Fig. 1, using DDF and its correspondence in Spark[3]. As the Fig. 4a shows, first we import our `COMPSsContext`, an internal DDF Library's abstraction, that is responsible for maintaining the context of a workflow using DDF objects, following the machine learning functions available in the library. From the operators supported by the API, we can read a "csv" file stored in HDFS, which returns a DDF object. After that, a flow can be created by using DDF operators, where each operator contains a previously implemented COMPSs function. The input and output of each operator is fixed: a function can have one or two data inputs, each a DDF variable, which internally keeps a list of n DataFrames. In addition, each operator has its particular parameters, described in the official documentation. The output of each function will be a DDF variable (*e.g.*, when the result is produced by the transformation of input data), a primitive data type (*e.g.*, the result of a statistical operation), or a simple DataFrame (*e.g.*, when the result is a table that can fit in memory). Internally, some machine learning algorithms may have more than one stage, which produce different types of output; however, the final output will follow the mentioned standardization. DDF can export the data to users that want to

[2]available at: `https://eubr-bigsea.github.io/Compss-Python`

[3]complete code is available on GitHub: `https://github.com/eubr-bigsea/Compss-Python/tree/master/tests/benchmark/titanic`

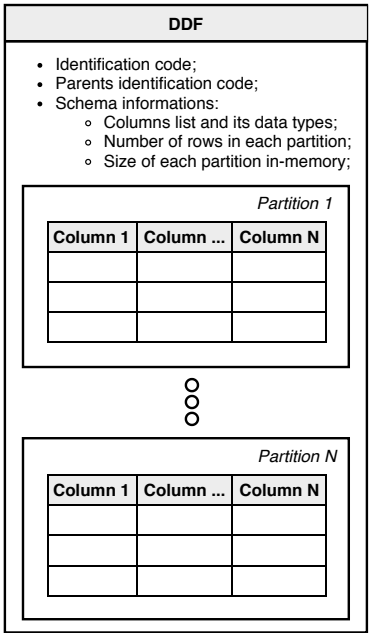

Figure 3: Internal structure of DDF.

use their own custom algorithms following the default COMPSs interface, and also import their DataFrame-based data to DDF.

5.1. Lazy evaluation and grouping tasks

The `COMPSsContext`, allows DDF to adopt lazy evaluation: when an operator is submitted, COMPSsContext adds that operation to a queue that describes the operation flow. Currently, operations in the queue are mapped to three types based on their dependency category: (*i*) operations with *narrow* dependencies are labeled *serial*, which indicates that they can be grouped with others if there is no bifurcation in their flow; (*ii*) operations that involve more than one processing stage are labeled as *last*, indicating an operator in which the first stage must be done individually (ending any existing chain of *serial* operators, and the second can be grouped with the following tasks, if they have a *serial* label (*e.g.*, the sort operation has two stages, the first one, where partitioning occurs, which cannot be grouped, and the second one, where the ordering itself takes place; the later stage can be grouped with other *serial* operations); and (*iii*) operations with *wide* dependencies are labeled *others* and indicate that they currently cannot benefit from optimization policies, like grouping tasks, and must be submitted individually. Similar to Spark, operations can be labeled as a transformation operation, that transforms a DDF into another one, or an action, that will force the execution of the flow. Transformation tasks are queued until an action (like save or cache) is submitted.

```
from ddf_library.context import COMPSsContext
from ddf_library.functions.ml.feature import
   StringIndexer, StandardScaler

compss = COMPSsContext()
...
ddf0 = compss.read\
 .csv('hdfs://namenode:9000/titanic.csv', schema=Schema)

sti = StringIndexer().fit(ddf0, input_col='Embarked')

ddf1 = sti.transform(ddf0, output_col='Embarked')\
 .drop(['PassengerId', 'Cabin', 'Ticket']) \
 .dropna(features, how='any')\
 .replace({'male': 1, 'female': 0}, subset=['Sex'])\
 .map(title_checker_udf, 'Name')\
 .map(age_categorizer_udf, 'Age')\
 .map(fare_categorizer_udf, 'Fare')




scaler = StandardScaler(with_mean=False, with_std=True)\
 .fit(ddf2, input_col=features)


ddf2 = scaler.transform(ddf2, output_col=features)\
 save.parquet('hdfs://namenode:9000/out')

compss.stop()
```

(a) Titanic's workflow in DDF

```
from pyspark.sql import SparkSession
from pyspark.ml.feature import StringIndexer,\
 VectorAssembler, StandardScaler

spark = SparkSession.builder.getOrCreate()
...
ddf0 = spark.read\
 .csv('hdfs://namenode:9000/titanic.csv',schema=Schema)

sti = StringIndexer(inputCol='Embarked',
                    outputCol='New_Embarked').fit(ddf0)

ddf1 = sti.transform(ddf0)
 .drop('PassengerId', 'Cabin', 'Ticket', 'Embarked')\
 .dropna(columns, how='any')\
 .withColumn('Sex', replace_udf(df.Sex))\
 .withColumn('Name', title_checker_udf(df.Name))\
 .withColumn('Age', age_categorizer_udf(df.Age))\
 .withColumn('Fare', fare_categorizer_uff(df.Fare))

ddf2a = VectorAssembler(inputCol=columns,
                        outputCol='features')\
 .transform(ddf1).drop(columns)

scaler = StandardScaler(withMean=False, withStd=True,
                        inputCol='features',
                        outputCol='features').fit(ddf2a)

ddf2b = scaler.transform(ddf2a)\
 .save.parquet('hdfs://namenode:9000/out.parquet')

spark.stop()
```

(b) Titanic's workflow in Spark

Figure 4: Comparison of running Titanic's workflow using DDF and Spark.

In Fig. 4 we divide the flow of operations using three variables (`ddf0` to `ddf2`) to match the color boxes in Fig. 1, representing how COMPSsContext will schedule those operations. For instance, operations related to the creation of `ddf1` can be submitted as a single task by COMPSsContext. However, we could write the code in different forms (*e.g.*, using a single sequence or multiple variables in different command lines); the abstraction is robust enough to analyze the flow of operations internally and decide if they should be grouped. Currently, `COMPSsContext` is capable of deciding how an operation should be submitted, whether is should be merged with others following it, or executed by itself. Its design and its Lazy evaluation nature support the addition of other techniques that can be added in the future like, for instance, re-organizing the order of some operations, when possible, to reduce the data size before a shuffle operation.

5.2. Serialization

In COMPSs, by definition, a task result is always stored on disk between tasks. When a task result is not a file, COMPSs serializes it by adopting a standard solution like Pickle [33] (for Python) to ensure compatibility of data types. Currently, in its architecture, the only intelligence added in this part is the choice of which method to use, one available through the built-in Python or using Numpy. However, because DDF handles structured data (*i.e.*, DataFrames), other serialization methods would be more suitable, as mentioned earlier. Because of this, we decided to take the responsibility of serializing a task result and choose the most suitable one. The COMPSs is flexible to allow this behavior by using an annotation to inform that the result is a file. Because DDF adopts a functional interface, all this is abstracted for the user.

We evaluated the main columnar file formats currently available in Python. Feather [34] is a format created early in the Arrow project as a proof of concept for fast, language-agnostic DataFrame storage for Python, however, has the limitation of not supporting storing columns that have Python lists. ORC files, although is efficient, only the reading method is available for the Python language. MessagePack [35], has shown good results, but its support has been removed from Pandas. The Apache Parquet, the chosen format, supports efficient compression and supports all DDF's data types.

After a task is finished, including the storage, the master node holds the location of that output and interprets it as a COMPSs *Future Object* until a synchronization is requested (which transfers that output to the master as data in memory). We use this feature in DDF to not overload the central computer: once a task has been executed, COMPSsContext updates its status to avoid recomputation and saves its result as a COMPSs *Future Object*. Besides the data output, each transformation on DDF also generates a schema output, as shown in Fig. 3. This schema contains some useful information about the current state, like the column name, the number of rows in each partition and its size in memory. This schema is a lightweight data used internally in many operations that need some previous information about the data without requiring auxiliary tasks — for example, the sample operation requires the length of each partition before it can define the sampling parameters.

5.3. Data locality and repartitioning

Operations with *wide* dependency tasks are expensive for task-based frameworks such as COMPSs, especially when their output can be large (*e.g.*, inner joins or sort operations). To minimize the data shuffle, when possible, DDF tries to reduce data size when partitioning (*e.g.*, the process of dropping duplicated rows involves a partial rows drop when data is being re-partitioned to reduce data size in the second step). However, this is not possible for all operations that need a shuffle; for instance, sorting is a process where the input size is equal of the output size. Unlike Spark, that manages it in-memory, COMPSs requires that each sub-fragment is written to disk to be transferred to workers that will be in charge of merging sub-fragments with same indices. Although partitioning can significantly reduce that overhead in COMPSs, it is still a high-cost step. When a task in COMPSs produces more than one output, all data are saved at the same time, at the end of the task, even if one output is produced at the beginning. A better approach, as Spark does, might be to save/transfer each output at the moment it is produced inside the task, reducing idle time.

Although COMPSs generates a DAG similar to Fig. 1b, which can be monitored by the user, the representation is at level of tasks. Because DDF encapsulates implementation details, we have made available a first version of a monitor *web UI*, complementary to COMPSs, which is at the level of operators, similar to Fig. 1a[4]. The monitor can be used to check the progress and obtain

[4] A sample of the actual COMPSs/DDF DAGs is available in GitHub along with the Titanic

some statistics of the execution. Interacting with the monitor, it is possible to verify, for instance, which tasks had their result persisted and which tasks have already been executed but had their results deleted.

6. Evaluation

The main purpose of this assessment is to validate DDF as a high-level abstraction capable of generating optimized PyCOMPSs code. We analyzed the performance gain of using our API in contrast to a traditional implementation that does not follow the guidelines mentioned in Section 4. In addition, we compared the performance of COMPSs using our data abstraction with equivalent Spark applications.

6.1. Experimental setup

In our evaluation, we compared DDF's performance to Apache Spark because that framework has similar characteristics to DDF, including functional interface, ETL and ML algorithms that are focused on big data, and other optimizations discussed in this paper.

We perform our experiments on a private cloud at Universidade Federal de Minas Gerais. All experiments used COMPSs (v. 2.6), HDFS (v. 3.1.3), or a Spark (v. 2.4.4) cluster with a dedicated master node and eight worker nodes. The virtualized machines had Intel E56xx processors of 2.5 GHz with 4 cores, 8 GB of RAM, with Ubuntu Linux 18.04 LTS. Although the number of available cores used in our experiment is small in comparison to usual Big Data clusters, this setup configuration represents a common pattern in many environments.

6.2. Selected applications

Big data applications are often based on ETL and Machine learning algorithms. Because such algorithms may have different properties, we selected six of them that cover some of the major execution behaviors in this context. Our goal was to cover common application patterns, like iteration and/or regularity. Thus, we select the following algorithms: Titanic workflow, KMeans, Distributed Support Vector Machines (SVM), Sort , Distinct and *People-Paths*.

(*i*) Titanic's workflow (as presented in Fig. 1) is a good example of a long flow of operations used in Big Data analytics for feature engineering. In this type of application, much work is done to extract features from raw data via ETL operations, to be later used in a Machine Learning model.

(*ii*) KMeans is a classical machine learning algorithm for data clustering, and it is a good example of an iterative, intensive application pattern. The goal of the algorithm is to classify a given data set into a certain number of clusters (K). Each cluster has a centroid. The algorithm works iteratively in such a way that in every iteration each data point is assigned to the nearest centroid.

application code

Those operations are compute-intensive tasks. The algorithm iterates until the centroids do not change their location or some other criterion is fulfilled.

(*iii*) SVM is a classical machine learning algorithm for data classification, and it is another good example of computationally intensive iterative application. The goal of this algorithm is to find the best hyper-plane that divides a dataset in two parts based on its labels. The algorithm works iteratively. In every iteration, each data point is used to measure the distance from that plane. The plane is refined at the end of the iteration. The algorithms stops when it reaches a maximum number of iterations or a threshold.

(*iv*) Sorting is a very common and useful data-intensive operation. The sort operation has a wide dependency, and because of that, it cannot be easily parallelized. It is an example of an operation that could be used a re-partition step (by range and by hash, respectively) to improve its performance.

(*v*) Distinct (drop duplicate rows) is other example of an operation with a wide dependency. Differently from Sort, it can also be optimized by trying to reduce the data set before the re-partitioning step.

(*vi*) *People Paths* [36] is an application for smart cities that performs a descriptive analysis on bus GPS and passenger ticketing data, finding paths taken by users of the city Public Transportation system during a time period. It then matches the paths origins/destinations with the city neighborhoods social indicators (*e.g.*, population, income and literacy rates). The application has four inputs: the buses ticketing and GPS data, the shapefile with the map of the city neighborhoods and the census data of the target city. Figure 5 shows the workflow for People Paths.

The first five applications are used with artificially generated input data: for Titanic, we replicated the original data set multiple times to create an input file varying from 2 to 20 GB; after the data interpretation in memory, that size varied from 8.8 GB to 94 GB; the other applications use artificial data generated by a uniform distribution varying from 10^8 to 10^9 rows. For KMeans, four columns are used as features (varying from 3 GB to 30 GB), and for SVM, one binary column is added to be used as label; for Sort, two of the four columns are used as keys to order; for Distinct, two of four columns were used to define if a row is equal to others, besides that, we consider two scenario, one with many duplicated keys (numbers were generated varying from 1 to 10^3) and other with few duplicated keys (in this case, number were generated by varying from 1 to 10^5). For People Paths we used data from the city of Curitiba, in Brazil, collected by the EUBra-BIGSEA Project [37] that will be discussed later.

6.3. Impact of grouping tasks

The first experiment, in Fig. 6, evaluates the impact of grouping multiple functions in a single task using Titanic's workflow. That application, as shown in Fig. 1, has some sets of functions (represented by same-color boxes) that can be reduced to fewer tasks. For instance, all green boxes can be merged into a single task. We measured the elapsed time that corresponds to the code snippet of green boxes and also measured the total time of the complete application. We

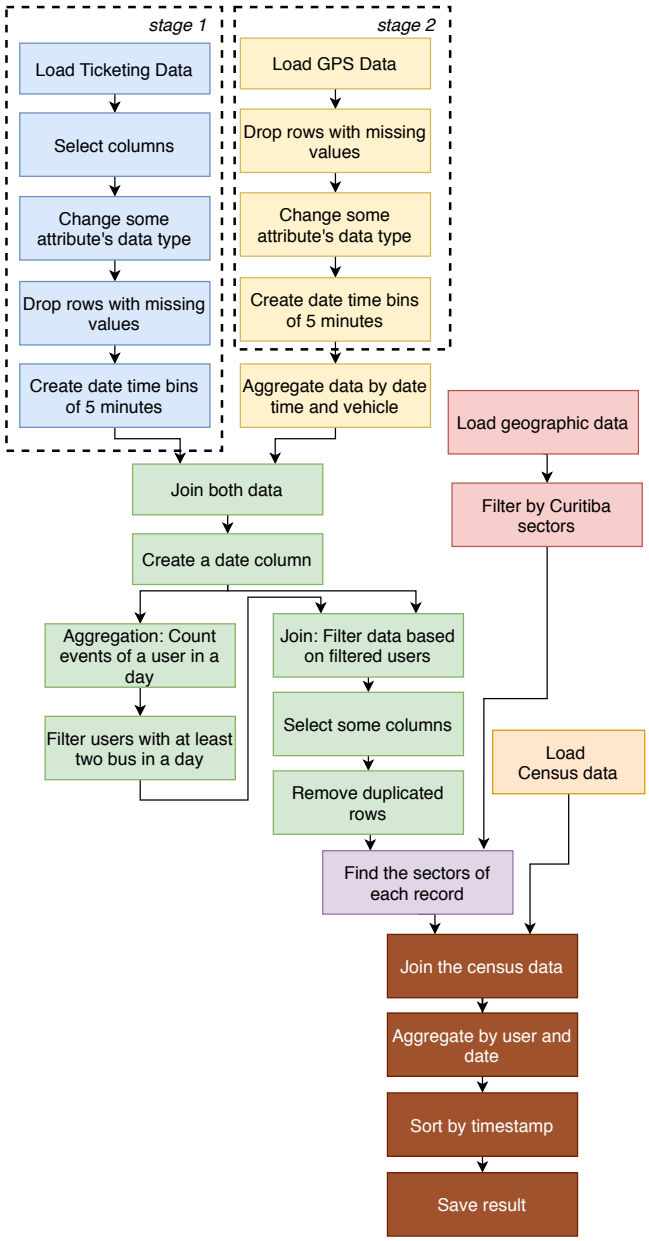

Figure 5: The People Paths's workflow.

computed the speedup achieved and its standard deviation [38] by using DDF against the original COMPSs code without that optimization. Results shown are the average of ten executions; the coefficient of variation of the results was

below 5% in all cases.

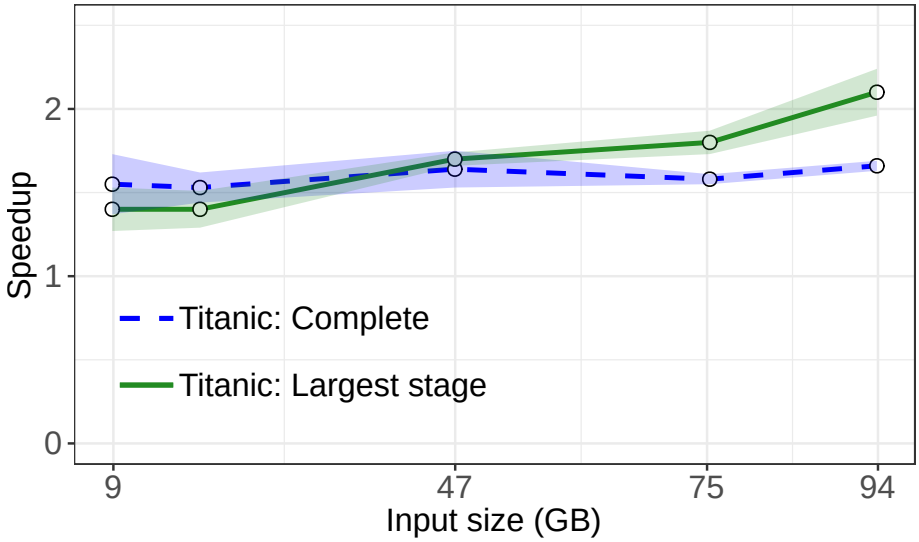

Figure 6: Speedup of grouping tasks on Titanic's workflow against traditional implementation.

The line in Fig. 6 identified as "Titanic: Largest stage" represents the speedup of just the group of operations in the green boxes in Fig. 1 when using DDF (that uses lazy evaluation to groups functions when possible) against a direct implementation in COMPSs without grouping tasks. The speedup for all input sizes considered is increasing, varying from 1.4 to 2.1, confirming the importance of using that technique. When we look at the complete application (blue line, "Titanic: Complete"), we have a speedup approximately constant of 1.6. One possible reason for this is that the complete application involves many tasks, some of which have wide dependencies, and also because the save operation is expensive (it involves saving data in HDFS, with replication factor 3); all this amortizes the speedup of the technique in this case.

6.4. Impact of a columnar serializer

We also used the same Titanic application to measure the impact of using the external DDF's serializer, which saves in Parquet format, taking that responsibility from COMPSs. In this experiment, we executed the application implemented in DDF (which groups functions when possible) and varying the serialization method by using the standard COMPSs way or by using the DDF alternative (which takes on COMPSs's serialization responsibility to be able to save in Parquet format). The speedup of this change on the execution time is shown in Fig. 7. The line identified as "Largest: columnar versus default" represents the speedup of the green boxes stage, which contains seven operations in the same stage, just like the previous experiment and saving the result in Parquet format. In this case, since it contains only one stage, the benefits of

using a more efficient serializer could not be noticed. However, as we deal with more complex workflows (as shown in "Complete: Columnar versus default"), more stages will be necessary, and so, the benefits of using a columnar format increase. In the range of the current experiment, the speedup increased to 1.4 for the data sizes tested.

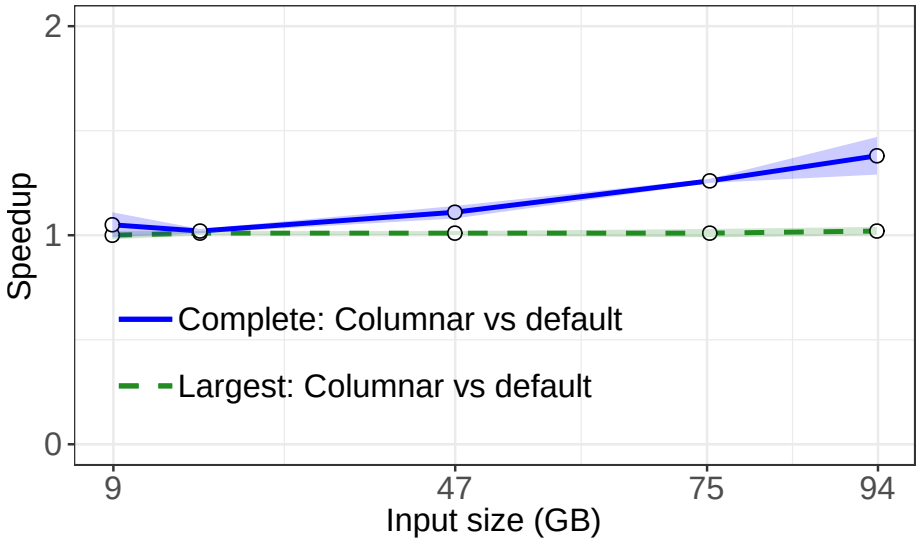

Figure 7: Columnar-based serializer speedup on Titanic's workflow against traditional COMPSs serializer.

6.5. Impact of (re-)partitioning data

Fig. 8 illustrates the impact of using a re-partitioning approach in operations that need a consensus among its fragments (as exemplified by Sort). In order to conduct this experiment, we compared the Sort operation implemented in DDF, which uses an approach of re-partitioning the data to be sorted by range values, to an implementation of Batcher odd-even mergesort [39], popular in GPU scenarios, where data is sorted in pairs following a priority sequence. We show visual traces of both executions created by the COMPSs runtime (Fig. 8(a) and 8(b), respectively). Each gray line in both traces represents a thread. Each worker node has five threads, one main thread that communicates with the master (represented by pink boxes) and other four threads to execute parallel tasks. The Batcher approach (Fig. 8(a)) is inefficient in big data scenarios[5], since it requires many steps and many transfers of partial results to other workers (red lines). Because there are many concurrent writes to disk, the serialization

[5]Despite its inefficiency in this case, among the possible versions of sort, the Batcher algorithm avoids sincronization steps that are known to cause overheads in COMPSs.

of results also takes a lot of time (green boxes). On the other hand, the DDF approach (Fig. 8b) is more efficient, with a speedup of 3.7 in this setup. The total time comprehends the definition of keys used to split data in new fragments, the splitting step itself, the time to merge fragments with same index and the time to sort data locally. In this case, the disk does not suffer an overload because there are fewer writer tasks and less serialization, leading to an execution time that is more related to the CPU time proper (white boxes).

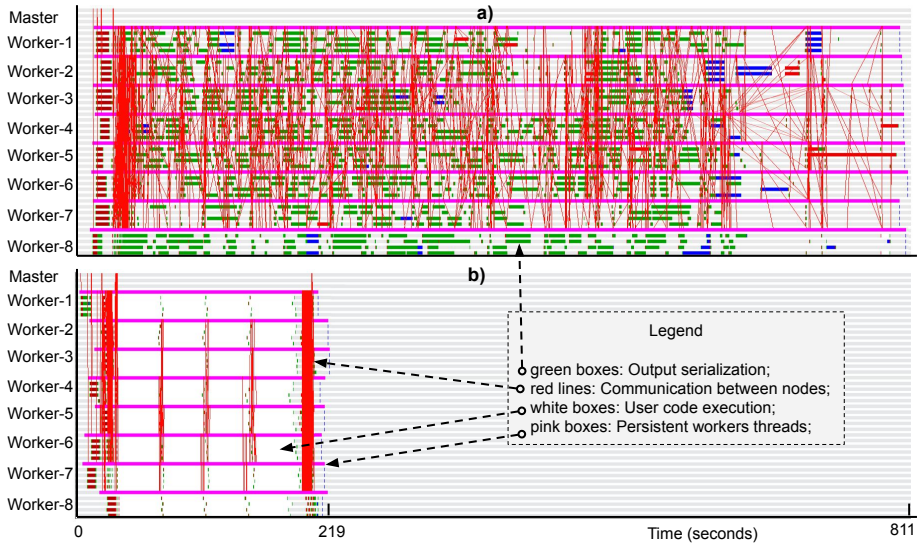

Figure 8: COMPSs Trace of two different Sort algorithms. a) Batcher approach (811 seconds); b) partitioning approach (219 seconds).

6.6. Performance comparison between frameworks

The next experiment compared some classes of algorithms and operations using DDF and Spark's DataFrame library. The results for Titanic application is shown in Fig. 9a, KMeans and SVM results are shown in Fig. 9b, Sort and Distinct results are shown in Fig. 9c. Table 1 shows results for People Paths, a real use case.

When DDF is compared with Spark using a large ETL application as the Titanic's workflow (Fig. 9a), the speedup of DDF over Spark is nearly constant by 1.7 for all data input size to the complete application. When we consider only the largest stage, as mentioned before, the speedup have similar behavior by a factor of 1.4×. It makes sense, because Spark already implements task grouping, so the difference in execution times is more related to differences of performance between both runtimes, not due to the serialization overhead. This result indicates that DDF is capable of working with multi-tasking stages as efficiently as Spark. We believe that shows that Group functions are a powerful optimization technique; by comparing DDF with a naive implementation in

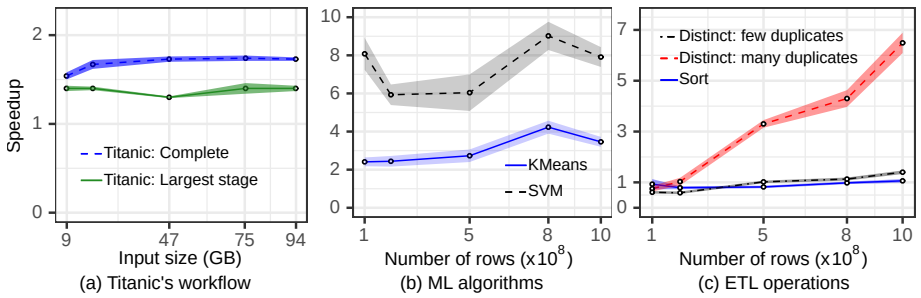

Figure 9: Speedup analysis of DDF in COMPSs applications against Spark implementations.

COMPSs (Fig. 6) we saw that the speedup depends on the data size and on the number of tasks that can be grouped. Also, by comparing with Spark, we saw that DDF can lead to optimized executions in COMPSs that are at least as good as Spark for Big Data applications.

Both DDF machine learning algorithms, KMeans and SVM (Fig. 9b), performed faster than Spark's versions. The speedup of DDF over Spark varied from 2.4 to 4.2 for the KMeans algorithm, and from 5.9 to 9.1 for SVM algorithm. The KMeans and SVM speedup indicated that DDF can surpass the Python language overhead by using a set of top libraries allied a set of well-designed implementations. Spark, on the other hand, performs the execution of these algorithms internally in Scala, following the same logic of its operators, that are responsible to define the computations and communication patterns, which have already been shown to be inefficient in ML/MLlib algorithms [40]. Unlike in DDF, the internal implementation of our algorithms is freer due to the COMPSs model, based on tasks, which allows the establishment of different types of communications and the use of external structures and libraries more easily.

In this set of experiments, Sort had the lowest speedup from 0.8 to 1.1 (Fig. 9c). As we saw in Fig. 8, the partitioning approach reduces the communication cost between nodes, but it is still an expensive step in COMPSs. During the execution of the split stage, each partial output waits until the end of its task to start the process of saving the output, creating some idle time. However, when we increase the input size, the speedup tends to increase until we get the same performance as Spark. That is probably because, as we increase the data size, Spark starts to have problems to keep data in memory, so it starts to serialize more data, as COMPSs does. Distinct applications have the same execution pattern of Sort, *i.e.*, require a re-partitioning to be able to check distinct elements. The difference, in this case, is that we can also apply a partial *distinct* function in each partition before the shuffle to reduce the data first. Because of that, we executed that operation in scenarios with many duplicate rows and others with few duplicates rows. As Fig. 9c shows, the optimization of apply partial functions in operations that are based on reduce data is a great technique. In a scenario with many duplicates rows, DDF was able to get a

N. of days	Ticketing data Size (MB)	Ticketing data N. records (K)	GPS data Size (GB)	GPS data N. records (K)	Speedup	Standard Deviation
5	173	982	1.5	15,403	1.23	0.02
10	397	2,260	3.6	37,124	1.42	0.04
15	639	3,646	5.7	58,647	1.28	0.03
20	895	5,108	7.7	79,490	1.36	0.06
25	1,144	6,534	9.8	101,115	1.43	0.07
30	1,525	8,708	11.9	122,549	1.39	0.04

Table 1: Speedup of DDF against Spark for People Paths application. Ticketing and GPS data records are shown in thousands.

speedup of 6.5× over Spark, indicating that the reordering technique is probably not done by Spark. However, in the case of Distinct with few duplicates rows, the speedup was smaller, varying from 0.6 to small data to 1.4 to large data, because a partial function is not able to reduce the data significantly in that scenario.

The People Paths application (as shown in Fig. 5) is an example of a real Big Data analytic application. In its logic, it involves many ETL operations with different execution patterns: operations with *narrow dependencies* that benefit from the technique of grouping tasks; operations with *wide dependencies* such as Join and Sort that benefit from partitioning; and operations such as Distinct (Remove duplicated rows), which can also be applied to a partial reduction. This application was implemented in Spark and DDF[6] and executed in a different number of data collection days, varying from 5 to 30 days, as shown in Table 1. The increase in the number of records for both data inputs is nearly linear as a function of the number of days. In Fig. 5 two of those stages are shown, the others were not represented in the image for simplification. Table 1 also contains the DDF speedup over Spark for a set of 10 runs each with its standard deviation. From the result, we can see that DDF has an average performance 35%, higher than Spark. In addition, we applied the Unpaired Two-Samples t-test on the largest of input data (30 days) to prove that the two systems are significantly different. The result of this test gives us a confidence interval (95 %) of (342.6, 377.4), meaning that DDF has a real performance between 343 to 377 seconds faster than Spark, for the evaluated set. In terms of speedup, this is equivalent to a speedup between 1.37 to 1.41. This shows us that the optimizations that DDF provides to COMPS users are a viable alternative. The same calculation is shown in Appendix A for the other experiments shown.

[6]available at: `https://github.com/eubr-bigsea/Compss-Python/tree/master/tests/benchmark/people_path/`

7. Conclusion

Despite the variety of distributed and parallel frameworks for HPC and Big Data, the use of functional-based programming interfaces is becoming a frequent model in many of them. In this paper, we discussed many implementation aspects that affect the performance of task-based frameworks, evaluated their impact on COMPSs and showed how a functional-based interface, a popular abstraction used in many Data Science tools, can be used to hide complexity in data-parallel algorithms, improving their performance. We explored the potential benefits of the integration between COMPSs, a powerful task-based framework originated in an HPC environment, with a functional-based interface. Although COMPSs has an easy-to-use native interface, the developer needs to take care of many implementation details to obtain the maximum of performance. With a functional-based API, many of those details can be hidden from the programmer.

We developed the DDF Library, a set of machine learning algorithms and operations on top of a functional-based DataFrame interface (DDF), in Python, for COMPSs. That interface implements a dynamic task evaluator capable of producing optimized code following the set of guidelines discussed in the paper. The COMPSs programming model has been shown to be very flexible, and those techniques and new ones can be added to DDF without major difficulties. We compared the performance of our proposed API with Spark and the results show that COMPSs with DDF is a high-performance, user-friendly solution for Big Data, with a large set of algorithms and operations that could be used as a viable programming environment. The fact that DDF/COMPSs outperforms Spark in an infrastructure usually related to Big Data, indicates that COMPSs is a versatile system.

In view of the results presented, we hope to help on this path of convergence with the contribution of DDF, and with the validation that such techniques can be successfully incorporated into an HPC system. The fact that DDF has been implemented in COMPSs allows it to be combined with the various HPC integration and supports that COMPSs already provides natively. DDF can be used by users already in the HPC world that are facing big data problems. But we also expect that this data abstraction helps users familiar with Big Data environments to use COMPSs as an easy alternative for a high-performance framework suited for Big Data applications.

The ongoing work includes developing the support for more functions in DDF and improving the optimization guidelines, for instance: to support the dynamic reorganization of tasks to reduce data volume during the shuffle step, by analyzing the memory footprint of each DDF partition that is already collected by DDF in the schema information; or to support logical optimizations, for example, by reorganizing code.

Experiment	Application	Speedup	95% CI
Impact of grouping tasks	Titanic: Largest stage	2.09	(1.74, 2.34)
	Titanic: Complete	1.66	(1.62, 1.68)
Impact of serialization	Largest: Columnar vs default	1.02	(1.01, 1.03)
	Complete: Columnar vs default	1.38	(1.30, 1.39)
Comparison of DDF vs Spark	Titanic: Largest stage	1.40	(1.38, 1.42)
	Titanic: Complete	1.73	(1.72, 1.74)
	KMeans	3.46	(3.38, 3.55)
	SVM	7.91	(7.79, 8.03)
	Sort	1.06	(1.02, 1.11)
	Distinct: many duplicates	6.49	(6.07, 6.90)
	Distinct: few duplicates	1.39	(1.36, 1.44)
	People Paths	1.39	(1.37, 1.41)

Table A.2: Speedup summarization for all experiments with the 95% confidence interval.

Acknowledgments

This work was partially supported by CAPES, CNPq, Fapemig and NIC.BR, and by projects Atmosphere (H2020-EU.2.1.1 777154) and INCT-Cyber.

Appendix A. Confidence interval study

Table A.2 summarizes the average speedup with respect to the largest data input scenarios for all experiments shown in this paper. For each experiment, we also applied an Unpaired Two-Samples T-Test to obtain a 95% confidence interval about the difference of execution time between the evaluated systems. In all scenarios, the results were statistically significant. Finally, we calculated a range of speedup based on the obtained interval (column 95% CI) by the equation $(1+c1/avg_a, 1+c2/avg_a)$, where: *c1* and *c2* are the confidence interval range by an Unpaired Two-Samples t-test and avg_a is the execution time mean of the main evaluated system.

References

[1] J. Dean, S. Ghemawat, Mapreduce: Simplified data processing on large clusters, in: Proceedings of the 6th Conference on Symposium on Operating Systems Design & Implementation - Volume 6, OSDI'04, USENIX Association, Berkeley, CA, USA, 2004, pp. 10–10.

[2] T. White, Hadoop: The Definitive Guide, 4th Edition, O'Reilly Media, Inc., Sebastopol, CA, USA, 2015.

[3] A. B. Yoo, M. A. Jette, M. Grondona, SLURM: Simple Linux Utility for Resource Management, in: D. Feitelson, L. Rudolph, U. Schwiegelshohn (Eds.), Job Scheduling Strategies for Parallel Processing, Springer Berlin Heidelberg, Berlin, Heidelberg, 2003, pp. 44–60.

[4] P. Schwan, Lustre: Building a file system for 1000-node clusters, in: Proceedings of the Linux symposium, Linux symposium, Ottawa, Ontario, Canada, 2003, pp. 380–386.

[5] G. Fox, et al., Big data, simulations and HPC convergence, in: Workshop on Big Data Benchmarks, Springer, 2015, pp. 3–17.

[6] M. Asch, 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, The International Journal of High Performance Computing Applications 32 (4) (2018) 435–479.

[7] E. Tejedor, et al., PyCOMPSs: Parallel computational workflows in Python, The International Journal of High Performance Computing Applications 31 (1) (2017) 66–82.

[8] P. Wickramasinghe, et al., Twister2:TSet high-performance iterative dataflow, in: International Conference on High Performance Big Data and Intelligent Systems (HPBD&IS 2019), IEEE, 2019, pp. 55–60.

[9] M. Zaharia, et al., Resilient distributed datasets: A fault-tolerant abstraction for in-memory cluster computing, in: Proc. of the 9th USENIX Symposium on Networked Systems Design and Implementation (NSDI 12), USENIX, San Jose, CA, 2012, pp. 15–28.

[10] P. Carbone, et al., Apache Flink: Stream and batch processing in a single engine, Bulletin of the IEEE Computer Society Technical Committee on Data Engineering 36 (4) (2015).

[11] S. Sehrish, J. Kowalkowski, M. Paterno, Exploring the performance of Spark for a scientific use case, in: IEEE International Parallel and Distributed Processing Symposium Workshops (IPDPSW), 2016, pp. 1653–1659.

[12] J. L. Reyes-Ortiz, L. Oneto, D. Anguita, Big data analytics in the cloud: Spark on Hadoop vs MPI/OpenMP on Beowulf, Procedia Computer Science 53 (2015) 121 – 130, iNNS Conference on Big Data 2015 Program San Francisco, CA, USA 8-10 August 2015.

[13] M. Wilde, et al., Swift: A language for distributed parallel scripting, Parallel Computing 37 (9) (2011) 633–652.

[14] F. Pedregosa, et al., Scikit-learn: Machine learning in Python, Journal of Machine Learning Research 12 (2011) 2825–2830.

[15] W. McKinney, pandas: a foundational python library for data analysis and statistics, Python for High Performance and Scientific Computing 14 (2011).

[16] M. Armbrust, et al., Spark SQL: Relational data processing in Spark, in: Proceedings of the 2015 ACM SIGMOD international conference on management of data, ACM, 2015, pp. 1383–1394.

[17] X. Meng, et al., Mllib: Machine learning in apache spark, J. Mach. Learn. Res. 17 (1) (2016) 1235–1241.

[18] S. Caíno-Lores, et al., Spark-DIY: A framework for interoperable spark operations with high performance block-based data models, in: 2018 IEEE/ACM 5th International Conference on Big Data Computing Applications and Technologies (BDCAT), 2018, pp. 1–10.

[19] BSC. COMPSs, Pycompss distributed data set: User manual, `http://compss.bsc.es/releases/compss/latest/docs/DDS_Manual.pdf`, last access: 2020-06-11 (2018).

[20] J. Álvarez Cid-Fuentes, et al., dislib: Large scale high performance machine learning in python, in: 2019 15th IEEE International Conference on eScience, Vol. 1, 2019, pp. 96–105.

[21] Dask Development Team, Dask: Library for dynamic task scheduling, `https://dask.org`, last access: 2020-06-01 (2016).

[22] Dask Development Team, Dask-ML, `https://ml.dask.org/`, last access: 2020-06-01 (2017).

[23] Dask Development Team, Joblib, `https://ml.dask.org/joblib.html/`, last access: 2020-06-01 (2017).

[24] W. Santos, 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), 2017, pp. 745–748.

[25] Y. Tang, TF. Learn: Tensorflow's high-level module for distributed machine learning, arXiv preprint arXiv:1612.04251 (12 2016).

[26] L. M. Ponce, et al., Extension of a task-based model to functional programming, in: 2019 31st International Symposium on Computer Architecture and High Performance Computing (SBAC-PAD), 2019, pp. 64–71.

[27] L. M. Ponce, et al., Upgrading a high performance computing environment for massive data processing, Journal of Internet Services and Applications 10 (19) (2019).

[28] Apache Spark, Tuning - Spark 2.4.5 Documentation, `https://spark.apache.org/docs/latest/tuning.html`, last access: 2020-06-01 (2020).

[29] S. Melnik, et al., Dremel: Interactive analysis of web-scale datasets, Proceedings of the VLDB Endowment 3 (1) (2010) 330–339.

[30] Apache ORC, Apache ORC: High-performance columnar storage for hadoop, http://orc.apache.org/, last access: 2020-06-01 (2020).

[31] T. Ivanov, M. Pergolesi, The impact of columnar file formats on SQL-on-hadoop engine performance: A study on ORC and Parquet, Concurrency and Computation: Practice and Experience 32 (5) (2020) e5523. doi:10.1002/cpe.5523.

[32] S. van der Walt, S. C. Colbert, G. Varoquaux, The numpy array: A structure for efficient numerical computation, Computing in Science & Engineering 13 (2) (2011) 22–30.

[33] Python Software Foundation, pickle - Python object serialization, https://docs.python.org/3/library/pickle.html, last access: 2020-06-01 (2020).

[34] Apache Arrow, Feather File Format, https://arrow.apache.org/docs/python/feather.html, last access: 2020-06-01 (2016).

[35] S. Furuhashi, MessagePack: It's like JSON. but fast and small, https://msgpack.org/, last access: 2020-06-01 (2019).

[36] N. Andrade, et al., D7. 3 - Toolbox for GES^3 data initial release, Tech. rep., EUBra-BIGSEA, https://www.eubra-bigsea.eu/sites/default/files/D7.3%20-%20Toolbox%20for%20GES%C2%B3%20Data%20Initial%20Release_v1.pdf, last access: 2020-06-13 (2017).

[37] A. S. Alic, et al., BIGSEA: A big data analytics platform for public transportation information, Future Generation Computer Systems 96 (2019) 243–269.

[38] J. S. Firoz, et al., The value of variance, in: Proceedings of the 7th ACM/SPEC on International Conference on Performance Engineering, ICPE'16, Association for Computing Machinery, New York, NY, USA, 2016, p. 287–295.

[39] K. E. Batcher, Sorting networks and their applications, in: Proceedings of Spring Joint Computer Conference, AFIPS '68 (Spring), ACM, New York, NY, USA, 1968, pp. 307–314.

[40] Z. Zhang, et al., MLlib*: Fast training of GLMs using Spark MLlib, in: 2019 IEEE 35th International Conference on Data Engineering (ICDE), IEEE, 2019, pp. 1778–1789.