Extension of a Task-based model to Functional programming

Lucas M. Ponce*, Daniele Lezzi†, Rosa M. Badia†‡ and Dorgival Guedes*
*Universidade Federal de Minas Gerais (UFMG), Belo Horizonte, Brazil
Email: {lucasmsp, dorgival}@dcc.ufmg.br
† Barcelona Supercomputing Center - Centro Nacional de Supercomputación (BSC-CNS), Barcelona, Spain
‡ Spanish National Research Council (CSIC), Barcelona, Spain
Email: {daniele.lezzi, rosa.m.badia}@bsc.es

***Abstract*—Recently, efforts have been made to bring together the areas of high-performance computing (HPC) and massive data processing (Big Data). Traditional HPC frameworks, like COMPSs, are mostly task-based, while popular big-data environments, like Spark, are based on functional programming principles. The earlier are know for their good performance for regular, matrix-based computations; on the other hand, for fine-grained, data-parallel workloads, the later has often been considered more successful. In this paper we present our experience with the integration of some dataflow techniques into COMPSs, a task-based framework, in an effort to bring together the best aspects of both worlds. We present our API, called DDF, which provides a new data abstraction that addresses the challenges of integrating Big Data application scenarios into COMPSs. 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 runtime. Besides the performance optimization it provides, the API facilitates the development of applications by experts in the application domain. In this paper we evaluate DDF's effectiveness by comparing the resulting programs to their original versions in COMPSs and Spark. The results show that DDF can improve COMPSs execution time and even outperform Spark in many use cases.**

***Index Terms*—COMPSs, Big Data, Performance Evaluation, DataFlow Programming**

I. Introduction

Convergence between high-performance computing (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 [1], [2]. Several proposals of higher-level abstractions have emerged to address the requirements of these two areas in computer systems [3], [4]. Recent frameworks, like COMPSs [3], Twister2 [4], Spark [5] and Flink [6], share a common dataflow programming model, but are still focused on a single 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 for users 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 that data is the output of a previous task, as it is the case in COMPSs, because it involves data serialization and deserialization steps. Because of that, a common practice adopted by advanced programmers is to minimize the number of different tasks by joining 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 join 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 [5], [6] 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 [7], [8] shows that, depending on the application (*e.g.*, matrix computations), those frameworks achieve good scalability, but poor performance when compared with an MPI implementation.

In that context, our work discusses our experience in bringing together the programming model of COMPSs and the functional programming abstractions usually found in frame-

DOI 10.1109/SBAC-PAD.2019.00023

works 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, that provides 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, and it has good performance [7].

To describe our work, the remainder of the paper is structured as follows: Section II presents some related work; Section III introduces the COMPSs framework and Section IV presents some optimization techniques; Section V presents our API, which provides a new data abstraction and interface to COMPSs users. The validation of our solution is discussed in Section VI, and Section VII presents our conclusions and discusses future work.

II. Related work

While dataflow is a prevalent model in many parallel and distributed programming frameworks [4], functional programming is slowly becoming a commom interface. In addition to Big Data frameworks like Spark, Flink and Swift [9], functional interfaces are also being frequently used in other Data Sciences programming tools (*e.g.*, Scikit-Learn[1] and Pandas[2] 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 [10]. 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 [11] 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 [12], 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 [4], DDS [13] and DisLib[3] 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 provide an abstraction similar to the RDD and a library like Spark's MLlib, respectively. Using the DDS interface, applications can be written using operators like `load`, `map`, `filter`, `and reduce` and using Dislib, users can execute machine learning algorithms. Although both projects are in Python, they do not have a strong integration to Spark, so users might need to convert their data to use both frameworks.

The DDF Library we present here resembles TSet and DDS by providing DDF, a new auxiliary data abstraction for COMPSs to handle Big Data. Unlike those projects, we adopted a DataFrame abstraction, an increasingly popular structure in Data Science [14], [15]. We implemented a large set of operations and algorithms, many of them not available in DDS and Dislib. 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.

III. The COMPSs framework

COMPSs is a programming 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 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.

[1]https://scikit-learn.org

[2]https://pandas.pydata.org

[3]https://github.com/bsc-wdc/dislib

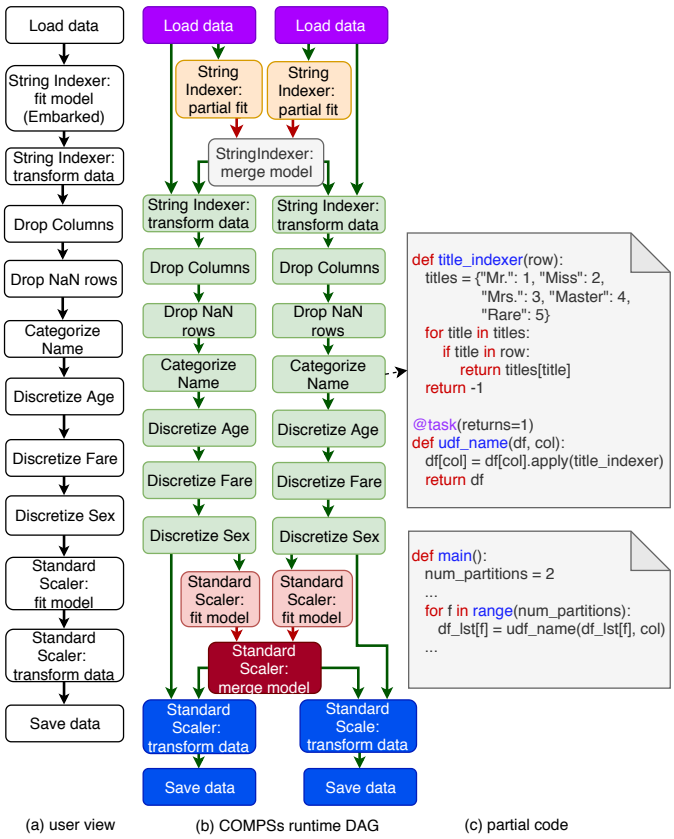

Fig. 1. Preprocessing Titanic's data set in PyCOMPSs.

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.

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. 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 [16]) to speed up the read step.

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[4]. 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 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, as `Categorize Name`. Others, however, like `String Indexer`, need more than one stage, since they have some actions that depend on the combination of 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.

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

IV. 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 interpreted (de-serialization) by the receiver.

Generic purpose frameworks, such as COMPSs and Spark, adopt standard serialization solutions to ensure universal compatibility with data that users will develop in their codes. Usually such frameworks 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.

A. Grouping tasks

The process to switch from one task to another can vary between different Dataflow-based frameworks. In COMPSs, for instance, the output of a task is always serialized and saved to disk until some other task requests it. In Spark, the result is serialized and saved in memory; only when that is not possible the result is written to disk. Whatever the procedure adopted, that context change always causes overhead. Building an efficient runtime depends on minimizing it by reducing the number of different tasks. To be able to do that, we must characterize the way tasks depend on each other. We define dependencies as *narrow* (green arrows in Fig. 1(b)) when each instance of a task depends on at most one instance of each of its parents; otherwise, it will be a *wide* dependency (red arrows in Fig. 1(b)). To filter or drop rows, and to replace values, are other examples of *narrow* dependencies; to sort data, to perform aggregations/joins and to find duplicated 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.

Advanced 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 in a single pipeline. For example, in the Titanic application, each set of tasks of equal color can be grouped into one set; in that example, the `Standard Scaler: transform data` and `Save data` task could be performed together. In case of 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.

B. Lazy evaluation

Frameworks generally create and analyze a DAG of code to decide when a set of tasks can be grouped. However, when it comes to an interactive environment, often used for Data Science exploratory tasks, the complete code is not always available. 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 an action (operations that return values to the user) is submitted, the enqueued tasks are executed. That technique allows frameworks to analyze and optimize the flow of operations.

C. 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 the 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 [17], where it re-organizes its data before the reduce step.

This idea can be also be extended by reducing the data before re-partitioning. For instance, the operation of removing duplicated rows based on their keys needs to shuffle 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 after an operation that reduces data. For instance, a flow that contains a sort operation following a filter. 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 this complexity of optimizing parallel code by analyzing the flow of the operations and re-organizing its tasks.

D. Exploring data locality

Besides the mentioned techniques, another way to decrease the transfers between nodes is by exploring data locality by scheduling tasks on nodes that already possess the input data. Recent frameworks, such as COMPSs and Spark, 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 [16]) 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 requested 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 executions to the best data provider.

E. 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 [3]. However, there are well-known factors (*e.g.*, the absence of strong typing), that can significantly decrease its performance. As a solution, many libraries such as NumPy[5], 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 connector in the Spark runtime.

V. The DDF Library

Based on our experience in developing Data Science applications and the approaches discussed in section IV, we developed the DDF Library, a high-level data abstraction for COMPSs applications with a functional language interface, and with 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 documented at the official site[6].

Distributed DataFrame (DDF) is based on the abstraction of 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. DDF runs

[5]https://www.numpy.org

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

on top of Pandas, a library providing high-performance, easy-to-use data structures and data analysis tools for Python. 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 IV before sorting. Also, some available algorithms in DDF use NumPy functions to speedup Python execution by using well-implemented C/C++ functions.

Fig. 2 shows the code of the Titanic application, previously mentioned in Fig. 1, using DDF[7]. As the figure shows, first we import our DDF data abstraction and the machine learning functions available in the library. From the operators supported by the API, we can read a file stored in HDFS. After that, a flow can be created by using other DDF operators. 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 generated 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 use their own custom algorithms following the default COMPSs interface, and also import their DataFrame-based data to DDF.

Internally, DDF implements `COMPSsContext`, a class to manage task submission based on the operation flow's context. It 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: operations with *narrow* dependencies are labeled *serial*, which indicates that they can be grouped with others if there is no bifurcation in their flow; operations labeled as *last* indicate operations that involve more than one processing stage, in which the first one must be done individually, and the second can be grouped with the following tasks, if they have a *serial* type (*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 operations with *wide* dependencies are labeled *others* and indicate that they currently cannot benefit from optimization policies, like

[7]the equivalent Spark code is available on GitHub: https://github.com/eubr-bigsea/Compss-Python/tree/master/tests/benchmark/titanic

```
from ddf_library.ddf import DDF
from ddf_library.functions.ml.feature import \
     StringIndexer, StandardScaler

...

ddf0 = DDF().load_text('/titanic.csv', storage='hdfs')

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

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, 'Name')\
    .map(age_categorizer, 'Age')\
    .map(fare_categorizer, 'Fare')

ddf2 = StandardScaler(with_mean=False, with_std=True)\
    .fit_transform(ddf2, input_col=features,
                   output_col=features)\
    .save("/titanic", storage='hdfs')
```

Fig. 2. Using DDF to implement Titanic's workflow.

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.

In Fig. 2 we divide the flow of operations into 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 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 will 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 aspects that can be implemented in the future like, for instance, re-organizing the order of some operations, when possible, to reduce the data size before a shuffle operation.

In COMPSs, by definition, a task result is always serialized and stored on disk between workers. 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 re-computation and saves its result as a COMPSs *Future Object*. Besides the data output, each transformation on DDF also generates a schema output. 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.

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. The next Section will evaluate this and other aspects of DDF performance.

VI. 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 gain of performance of using our API in contrast to a traditional implementation that does not follow the guidelines mentioned in Section IV. In addition, we have compared the performance of COMPSs using our data abstraction with Spark applications.

The applications used to evaluate the performance are the Titanic workflow, K-Means, Distributed SVM, Sort and Distinct. Titanic's workflow (as presented in Fig. 1) is a good example of a long flow of operations used in Big Data analytics. We choose K-Means and Distributed SVM as examples of computationally intensive iterative applications also used in many high-performance evaluations [4], [8]. Sort and Distinct (drop duplicate rows) are examples of wide dependency operations that need a re-partition step (by range and by hash, respectively). Distinct, differently from Sort, can also be optimized by trying to reduce data before the re-partitioning step. All input data are artificially generated: in 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 to 94 GB; the other applications use artificial data generated by a uniform distribution varying from 10^8 to 10^9 rows. In KMeans, four columns are used as features (varying from 3 to 30 GB) and in SVM, one binary column is added to be used as label; in Sort, two of the four columns are used as keys to order; in Distinct, the four columns are used to define if a row is equal to others. We perform our experiments on a private cloud at Universidade Federal de Minas Gerais. All experiments used COMPSs (v. 2.4), HDFS (v. 2.7), or a Spark (v. 2.4) 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 16.04 LTS.

A. Impact of grouping tasks

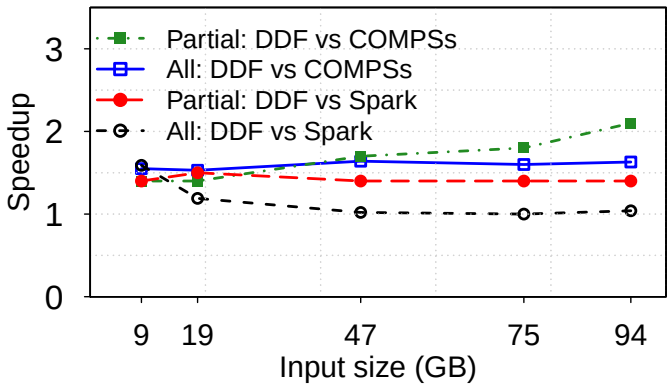

Fig. 3. DDF speedup on Titanic's workflow agains traditional COMPSs and Spark implementations. *All* indicates speedup of the whole application; *Partial* considers just the time of specific task code.

The first experiment, in Fig. 3, 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 fewer tasks. For instance, all green boxes in can be merged into a single task. We measured the elapsed time that comprehends the code snippet of green color boxes and also measured the total time of the complete application. We computed the speedup achieved by using DDF against the original COMPSs code, without that optimization, and also the DDF speedup over Spark. Results shown are the average of ten executions.

The line in Fig. 3 identified as "Partial: DDF vs COMPSs" represents the speedup of just the group of operations in the green boxes in Fig. 1 when using DDF (which 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, "All: DDF vs COMPSs"), 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.

When DDF is compared with Spark, the speedup of "Partial: DDF vs Spark" is also nearly constant (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. However, when comparing the complete application, the speedup is close to 1.5 when we use a small data set but,

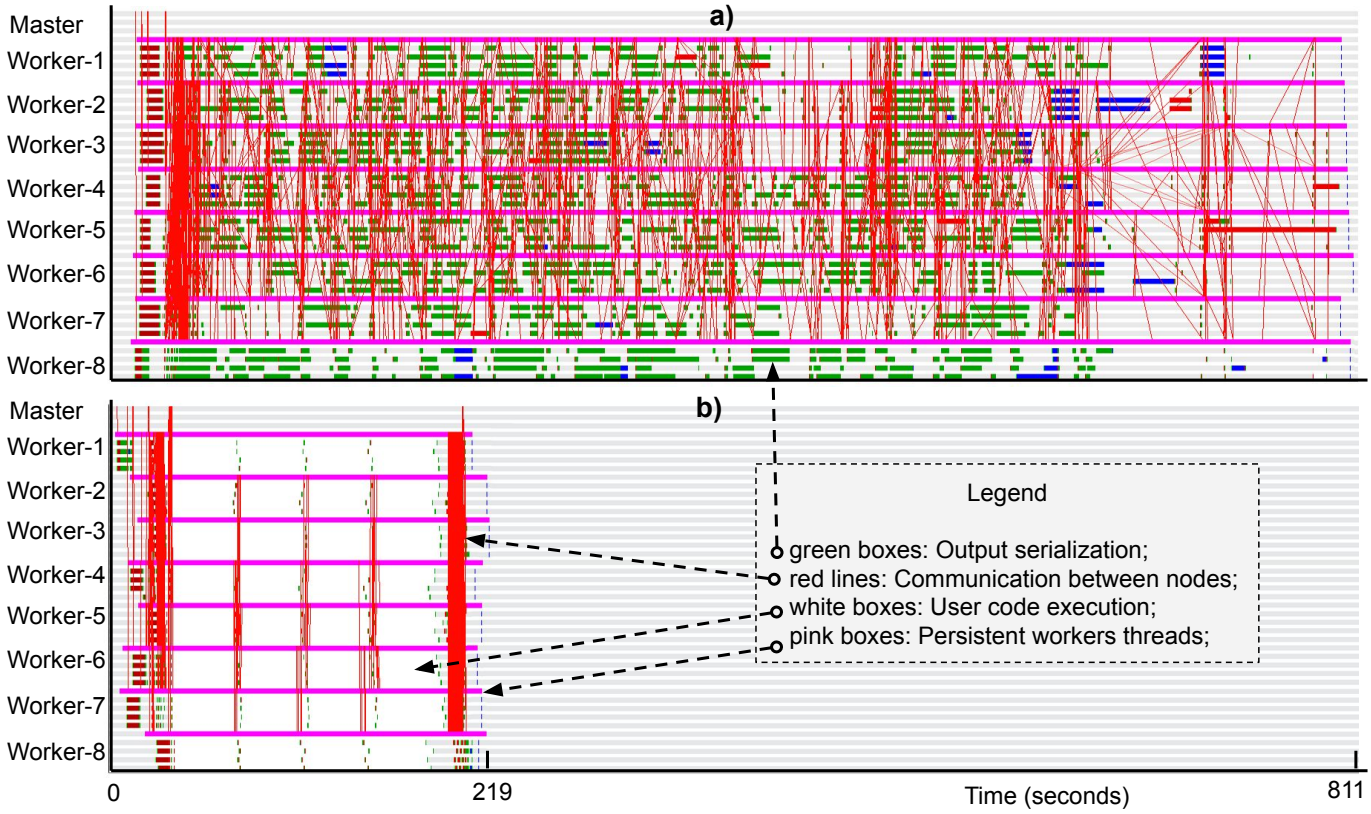

Fig. 4. COMPSs Trace of two different Sort algorithms. a) Batcher approach (811 seconds); b) partitioning approach (219 seconds).

as we increase data size, the costs of the data serializations that could not be avoided in COMPSs are greater then the benefits of DDF, and the speedup got reduced and stabilized close to 1. We believe that shows that Group functions are a powerful optimization technique; by comparing DDF with a naive implementation in COMPSs 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, what is a positive result for COMPSs.

B. Impact of (re-)partitioning data

Fig. 4 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 [18], 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. 4(a) and 4(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. 4(a)) is inefficient in big data scenarios, 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 of results also takes a lot of time (green boxes). On the other hand, the DDF approach (Fig. 4b) 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).

C. Performance comparison between frameworks

The next experiment (Fig. 5) compared some classes of algorithms and operations using DDF and Spark's DataFrame library. Both DDF machine learning algorithms, KMeans and SVM, performed faster than Spark's versions. Initially, the Spark version of KMeans algorithm performed slightly faster than DDF. However, as we increased the number of rows, the DDF speedup over Spark increased until 2.4. In SVM, DDF was superior in all cases. In this experiment, Sort had the lowest speedup (from 0.8 to 1.0). As we saw in Fig. 4, the partitioning approach reduces the communication cost between nodes, but it is yet a 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

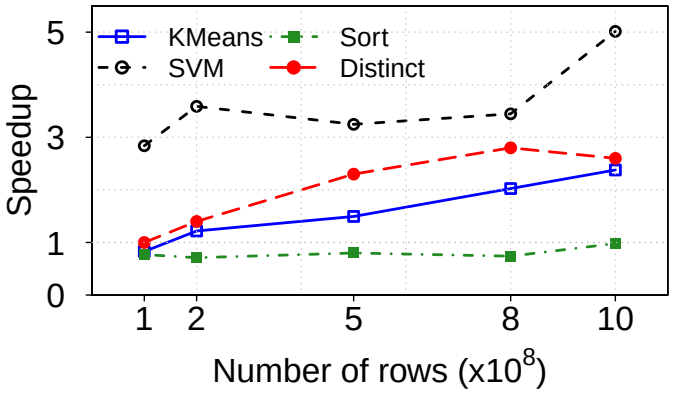

Fig. 5. DDF speedup over Spark for algorithms KMeans and SVM, and sort and distinct operations.

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. On the other hand, Distinct is faster because it can be optimized by performing the partial drop of duplicate rows in initial partitions to reduce data shuffle in the later stages, what is probably not done by Spark.

VII. 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 the COMPSs system 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. This interface implements a dynamic task evaluator capable of generating optimized code following a set of guidelines discussed in Section IV. 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. We 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.

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.

References

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

[2] 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*, vol. 32, no. 4, pp. 435–479, 2018.

[3] E. Tejedor *et al.*, "PyCOMPSs: Parallel computational workflows in Python," *The International Journal of High Performance Computing Applications*, vol. 31, no. 1, pp. 66–82, 2017.

[4] P. Wickramasinghe *et al.*, "Twister2:TSet high-performance iterative dataflow," in *International Conference on High Performance Big Data and Intelligent Systems (HPBD&IS 2019)*, no. 6. IEEE, 2019.

[5] 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)*. San Jose, CA: USENIX, 2012, pp. 15–28.

[6] 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*, vol. 36, no. 4, 2015.

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

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

[9] M. Wilde *et al.*, "Swift: A language for distributed parallel scripting," *Parallel Computing*, vol. 37, no. 9, pp. 633–652, 9 2011.

[10] 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.

[11] X. Meng *et al.*, "Mllib: Machine learning in apache spark," *J. Mach. Learn. Res.*, vol. 17, no. 1, pp. 1235–1241, Jan. 2016. [Online]. Available: http://dl.acm.org/citation.cfm?id=2946645.2946679

[12] 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)*, Dec 2018, pp. 1–10.

[13] BSC. COMPSs, *PyCOMPSs Distributed Data Set: User Manual*, available in: http://compss.bsc.es/releases/compss/latest/docs/DDS_Manual.pdf. Last access: 2019-06-14.

[14] 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.

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

[16] L. M. Ponce *et al.*, "Extensão de um ambiente de computação de alto desempenho para o processamento de dados massivos," in *Simpósio Brasileiro de Redes de Computadores (SBRC)*, vol. 36, 2018.

[17] J. Dean and S. Ghemawat, "Mapreduce: Simplified data processing on large clusters," in *Proceedings of the 6th Conference on Symposium on Operating Systems Design & Implementation - Volume 6*, ser. OSDI'04. Berkeley, CA, USA: USENIX Association, 2004, pp. 10–10. [Online]. Available: http://dl.acm.org/citation.cfm?id=1251254.1251264

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