Task-based programming in COMPSs to converge from HPC to Big Data

Javier Conejero*, Sandra Corella*, Rosa M Badia†, and Jesus Labarta*

* Barcelona Supercomputing Center (BSC)
C/ Jordi Girona, 31
08034 Barcelona, Spain
{francisco.conejero, sandra.corella, rosa.badia, jesus.labarta}@bsc.es
† Institut d’Investigació en Intel·ligència Artificial - Consejo Superior de Investigaciones Científicas (IIIA - CSIC), Spain

Abstract—Task-based programming has proven to be a suitable model for high-performance computing (HPC) applications. Different implementations have been good demonstrators of this fact, and have promoted the acceptance of task-based programming in the OpenMP standard.

Furthermore, in recent years, Apache Spark has gained wide popularity in business and research environments as a programming model for addressing emerging Big-Data problems. COMP Superscalar (COMPSs) is a task-based environment that tackles distributed computing (including Clouds), and is a good alternative for a task-based programming model for Big Data applications.

This paper describes why we consider that task-based programming models are a good approach for Big Data applications. The paper includes a comparison of Spark and COMPSs in terms of architecture, programming model and performance. It focuses on the differences that both frameworks have in structural terms, on their programmability interface, and in terms of their efficiency by means of three widely known benchmarking kernels: Wordcount, Kmeans and Terasort. These kernels enable the evaluation of the more important functionalities of both programming models and analyse different workflows and conditions.

The main results achieved from this comparison are: (1) COMPSs is able to extract the inherent parallelism from the user code with minimal coding effort as opposed to Spark, which requires the existing algorithms to be adapted and rewritten by explicitly using their pre-defined functions; (2) it is an improvement in terms of performance when compared with Spark, and (3) COMPSs has shown to scale better than Spark in most cases.

Finally, we discuss the advantages and disadvantages of both frameworks, highlighting the differences that make them unique, thereby helping to choose the right framework for each particular objective.

Keywords—Programming models; Distributed computing; Framework comparison; Big Data programming;

I. INTRODUCTION

One of the most important current research topics is the efficient exploitation of huge distributed environments, such as Datacenters and Clouds, in order to tackle the new challenges arising from the present era of Big Data. These efforts are mainly driven by research and business communities and are aimed at reducing the problem-solving time, achieving the efficient usage of the available resources, and producing effective programming models. An additional challenge involved in the efficient programming of Big Data applications is the need for convergence that is now required between the Exascale and Big Data worlds [21], [30]. However, this challenge depends on many factors that range from code definition, the computing framework and hardware capabilities.

A variety of frameworks currently exist that are aimed at exploiting distributed environments in an efficient way. Among these frameworks, Hadoop [1] and Spark [2] have attracted the most interest from the general public; Spark in particular is becoming very popular. While these two programming models focus on the definition of operators, COMPSs appears as an alternative based on the task–based programming paradigm.

COMP Superscalar (COMPSs) [4] is a task–based programming model designed to facilitate the development of applications for distributed infrastructures, such as Clusters, Grids and Clouds. To this end, COMPSs features a runtime system for exploiting the inherent parallelism of applications at execution time, thereby enabling the execution of sequential applications into distributed environments leveraging the coding effort.

Furthermore, Spark claims to be a “fast and general engine for large–scale data processing” [2]. It has aroused a considerable community interest and attracted a number of contributors focused on its improvement and exploitation for a large number of different projects and objectives. This constitutes one of the key advantages of Spark and has resulted in an increasing ecosystem that makes it attractive for a larger public.

Both models are designed to tackle the same problem while adopting a different approach. More specifically, Spark exploits the parallelism of its given operators, while COMPSs analyses and exploits the inherent parallelism of the user application. As shown in this paper, these features are relevant to the overall application development/migration and to performance.

The contribution of this paper is that it demonstrates how COMPSs provides a good alternative for programming Big Data applications, and also provides a comparison of both COMPSs and Spark frameworks in order to show their advantages and disadvantages. Since there are clear
differences between Spark and COMPSs, in this article they are analysed in detail according to their:

- Architecture: internal structure and components.
- Programming model: programmability and parallelization mechanisms.
- Performance: through the execution of three known benchmarking applications: Wordcount, Kmeans and Terasort, within the MareNostrum supercomputer.

There are several aspects that need to be considered in order to perform a successful application parallelization and execute it efficiently within distributed environments, such as programmability and code migration ease, performance tuning, adaptability and infrastructure management. By taking these aspects into account, this paper demonstrates that COMPSs is a good alternative for achieving this purpose.

With this objective and motivation in mind, we also present a discussion on the advantages and disadvantages of these models, highlighting the differences that make them unique and thus help to choose the right framework for each particular objective.

This article is structured as follows: Section II presents a brief overview of the current state of the art. Then, in Section III, COMPSs and Spark are presented with a focus on their architectures and programming models, as well as a discussion on their similarities and differences in these areas. Section IV provides the applications and performance comparison, with an evaluation of three well–known applications widely used for benchmarking distributed computing frameworks: Wordcount, Kmeans and Terasort. Finally, in Section V, the conclusions and future work are presented.

II. RELATED WORK

Given the high degree of interest in determining and improving the performance of distributed computing frameworks, this topic is being actively evaluated. Consequently, an extensive number of works exist that are aimed at providing thorough evaluations. In this sense, Spark has gained a significant reputation due to its improved performance when compared with its main competitors.

One of the first of these works that needs to be considered [36] is by Zaharia et al., who are the creators of Spark. In their article, the authors present Spark and provide a brief definition of the framework. They also present its programming model, with examples, its implementation and finally a short performance comparison between Spark and Hadoop.

Other works, such as [32], focus on the comparison of specific mechanisms in Spark and Hadoop for large scale data analytics. In this particular article, the authors center their attention on the map reduce implementation, and perform a thorough evaluation of the implementation differences and a final evaluation into a real cluster. This comparison uses the same benchmarking algorithms as our work: Wordcount, Kmeans and Sort, but including PageRank.

In [22], a comparison of Spark and Hadoop by using the Kmeans algorithm is presented. This short work analyses the differences in implementation and the performance results achieved. The authors claim that Spark will overtake Hadoop, although the scope of their experiments is very small (in terms of node and dataset sizes).

In [20], the authors compare three frameworks: the Message Passing Interface (MPI), Spark and Flink. One of the examples used is KMeans as in this work, although the problem sizes used herein are larger than the ones used in the above-mentioned work. Moreover, the computing platform used for their experiments is slightly smaller. The results they obtain indicates that Spark and Flink are less efficient than MPI due to the sub-optimal implementation of the reductions in those frameworks.

Moreover, extremely thorough works exist that conduct deep comparisons between distributed frameworks, such as [28]. In this master thesis, the author provides a comprehensive analysis of Hadoop and Spark.

Alternatively, there are works that extend these comparisons and include new and emerging distributed computing frameworks, such as Storm. For example, [19] presents a survey on Hadoop, Spark and Storm. This survey explains each framework in detail, focusing on their architecture and mechanism behaviour. The authors conduct a discussion on these topics from a theoretical point of view in order to identify the scenarios in which each one would fit the best, finally arriving at the conclusion that Hadoop will prevail. This work and the discussion it contains lack any performance comparison, which in some cases can be crucial.

There is a further work [31] that compares Spark on Hadoop vs MPI/OpenMP on Beowulf. This article uses two supervised learning algorithms, KNN and SVM-Pegasos, for the vertical and horizontal scalability evaluation, with a subset of 500,000 examples (represent 7GB) of the Higgs Data Set [17]. The decision of this small dataset is based on the objective of performing all the computation in memory. Unlike the previous works described in this Section, the authors evaluate their experiments in a larger distributed environment through the use of Google Cloud Platform (GCP [16]) with standard machines, but with different configurations (5, 10 and 20 machines, with 4, 8 and 16 cores each). The results show that MPI/OpenMP achieves a better performance under the defined conditions, although the authors remark on some of the benefits of using Spark, such as a distributed file-system with data replication management, hot node addition and ecosystem.

III. FRAMEWORKS

This section introduces the frameworks compared in this work: COMPSs and Spark. They are described in detail, with particular attention given to their architecture and
programming model, as a basis for a discussion at the end of the section.

A. COMPSs

COMPSs [18] is a programming framework aimed at enabling environment unaware Java, Python and C applications to be executed in distributed environments with minimal effort. To this end, COMPSs implements a task-based programming model, and while applications are written according to the sequential paradigm, the runtime is able to exploit their inherent parallelism, respecting the original behaviour by detecting and taking care of the data dependencies that may exist between tasks. Consequently, COMPSs is able to exploit the existing inherent parallelism at task level and deal with it across a distributed infrastructure, such as Grids, Clusters and Clouds, empowering interoperability.

COMPSs is able to provide scaling and elasticity features through Cloud virtual machines by adapting the number of available resources to the actual needs of the execution. This is achieved through the use of connectors for Infrastructure-as-a-Service (IaaS) offerings (e.g. Amazon EC2 [13] and for Open Cloud Computing Interface (OCCI) [14] compliant middlewares (like OpenNebula [3], [27] and OpenStack [15]).

The COMPSs programming model is designed to avoid modifications of the original application as possible. Consequently, its impact on the application has been reduced to a minimum, which means that the only requirement is to implement an interface description of the functions to be considered as tasks for Java/C/C++ applications and by the use of decorators [5] for Python applications. The interface does not interfere with the sequential execution of the application, so the application is ready to be executed in sequential and in distributed mode without the need of any modification.

Among the functionalities implemented by COMPSs, there is data dependency analysis, task scheduling, data transfer and fault tolerance.

1) Programming model: In this model, the user is responsible for identifying the functions to be executed as asynchronous parallel tasks, and for annotating them into an interface for Java/C/C++ applications or with a decorator into Python applications. The functions identified as tasks will be spawned into the available resources, thereby enforcing the data dependencies that may exist among them.

Figure 1 provides an example of the COMPSs programming model written in Python. This example represents the Wordcount application, in which two functions have been identified as tasks (with the decorator @task): (i) word_count and (ii) reduce_count. They are invoked for each block within the Data list. First, the word_count, which has one parameter (each block, input by default) and an output (a dictionary that contains all the appearances of each word within the block). Secondly, the reduce_count function, which has been defined to merge the partial results; it has two parameters (two dictionaries, where the first is defined as INOUT for the partial result accumulation). Finally, when the loop has finished iterating over the Data list, it a synchronization point requesting the value of the final result has been defined (through invocation of the compss_wait_on Application Programming Interface (API) function).

At execution time, the COMPSs runtime detects that there is a dependency between the output of each word_count and the input of each reduce_count, and also another between each reduce_count as a result of the INOUT direction of its parameter. For a very small data set were Data has four
blocks, the dependency graph would be the one depicted in Figure 1.

It is very important to note, that this code can be executed sequentially without COMPSs, or in parallel by executing it with the COMPSs runtime. This obviates the need to rewrite existing sequential applications when targeting parallel and distributed platforms.

A more extended description of the Java/C/C++/Python COMPSs syntax can be found in [34] and in the COMPSs manuals [4].

2) Architecture: Although the COMPSs architecture is complex, it can be simplified into three main layers (Figure 2). The most important, namely COMPSs runtime, implements the main runtime core functionalities. It is written in Java and performs the entire management of the application and environment. Consequently, Java applications can be executed directly on top of the runtime and processed through its customized loader which, given an application, manages its execution on the resources (which can be of different nature, such as Clouds or Clusters). During the execution, the runtime distributes the tasks over the resources, exploiting the inherent parallelism described in the task graph and enforcing the data dependencies in order to ensure the correctness of the application.

The layer Binding-commons lies over the COMPSs runtime layer and its objective is to enable the functionalities of the runtime to other languages. To this end, it has been designed as an API with a set of defined functions, written in C, and performs the communication with the runtime through the JNI [24], [7].

Finally, the top layer is composed of various bindings. For instance, a Python and a C/C++ binding, which enable the execution of Python and C/C++ applications with the COMPSs runtime.

COMPSs deployment over any infrastructure is based on the master-worker paradigm. From a pool of nodes assigned for the execution of an application, the first node acts as master node and is responsible for the management of the application, the data transfers, scheduling and infrastructure management. All the other nodes act as workers and are in charge of performing any action that the master requests them to do, such as executing tasks. The workers also have the ability to request data from other workers in order to satisfy any data dependency that may exist within the tasks to be performed.

By default, COMPSs works standalone with local storage, but it also provides support for interaction with existing storage technologies, such as dataClay [25] and Hecuba [34], through a simple API. In addition to the exploitation of Grids, Clusters and Cloud environments, COMPSs contains specific connectors to enable its deployment in Dockers, and is currently working on the integration with Mesos.

A final remark about the COMPSs runtime is that it has been instrumented with Extrae [6] in order to generate post-mortem runtime trace files that capture the runtime behaviour. As a result, the runtime behaviour can be analysed using Paraver [29], [8], which enables developers to understand how the application has been managed across the infrastructure, conducting performance analysis and detecting bottlenecks.

B. Apache Spark

On the other hand, Spark is a distributed in-memory data processing system that has gained greater popularity than Hadoop in the computing community. Its main objective is to provide a set of directives that users can employ in their applications, which internally are optimized to be executed in distributed environments through the Spark runtime. In order to make use of these directives, the data to be used within Spark applications must be mapped to Resilient Data Descriptions (RDDs), which represent immutable sets of data, and must be defined and respected by users for particular data structures (e.g. partitioning).

As a result from its wide popularity, a rich ecosystem surrounds it, in which the libraries must be highlighted (e.g. Machine Learning lib (MLlib), GraphX, Streaming and SparkSQL).

Among the functionalities implemented by Spark we find the following: task scheduling, data transfer, fault tolerance and checkpointing.

1) Programming model: At a high level, a Spark application consists of a driver program that runs the user’s main function and executes the defined parallel operations on a cluster.
To this end, the Spark programming model revolves around the RDD concept (Figure 3). Conceptually, RDD is the main logical data unit in Spark. An RDD is a distributed collection of objects, in the sense that each RDD is divided into multiple partitions. Each partition can reside in memory or on disk within a distributed infrastructure. Only two types of operations – transformations and actions – can be applied to the RDDs. The transformations are designed to perform an operation on the RDDs, which involves the creation of a new dataset (e.g. map, reduceByKey, etc.), while the actions are designed to execute a query to the RDDs (e.g. count, etc.). The complete list of transformations and actions is provided in the Spark documentation.

An example of the well-known Wordcount application for Spark is depicted in Figure 3. The application starts loading the data into an RDD (file), to which some transformations will be applied immediately after. In this particular application, since the objective is to count the number of appearances of each word in the data, in order to function one initial transformation (flatMap) will be applied to all RDD partitions, which will split the data considering the space as a word delimiter. This transformation will produce a new RDD, to which the next transformation (map) will be applied. The map transformation also contains a function to be applied to all partitions, which for the purpose of the application will assign the value 1 to all words with a Key-value structure. Finally, with the map results, the final transformation then is applied: reduceByKey. This transformation returns another dataset of key-value pairs where the values for each key are aggregated using the add function. Finally, the results of these three transformations are collected (by calling the collect action over the counts RDD) and saved to disk thanks to the saveAsTextFile action.

With these operations, Spark is able to manage the parallelization of the execution, taking into account the dependencies that exist between operations. Thus, it is capable of performing optimizations for particular sequences of operations, such as triggering certain operations (e.g. shuffle) in order to improve the overall performance and reduce the memory/storage footprint.

Spark provides a set of mechanisms for transforming raw data (e.g. plain text) into RDDs. However, these mechanisms may not be suitable for user-specific data structures, where the user will be required to override the data partitioning mechanism and implement it in accordance with the peculiarities of the data structure.

One of the most salient capabilities of Spark is the RDD fault-tolerant persistence, commonly known as caching. This persistence is designed to keep partitions in memory between operations in order to enhance future performance. However, the main memory is finite, so when no more partitions can fit into memory, Spark starts serializing and storing them into a slower storage level, such as disk. Spark automatically monitors the cache usage and tries to remove the unused partitions through a Least Recently Used (LRU) policy.

2) Architecture: Spark architecture can also be simplified into layers (Figure 4) and has a similar architecture to COMPSs. It consists of a main module, namely Apache Spark, which implements the runtime features. The runtime is in charge of managing the application execution and delegating the tasks into the underlying resources, as well as managing them in order to keep data consistency and transfer.

Spark provides support for Java and Scala as well as for Python applications. This binding, known as PySpark, lies on top of the Spark runtime. It implements all the supported operations and transformations, as well as carrying out the interaction between languages in order to interact with the Spark runtime.

Moreover, given the wide adoption of Spark, a large
number of libraries have been developed and thus provide a wide range of multi-disciplinary functionalities already implemented for Spark (e.g. MLlib, GrapX, Streaming and SparkSQL). One example of these libraries is MLlib, which is currently gaining wide popularity; it implements a set of machine learning algorithms, such as Kmeans, in order to facilitate the development of Spark applications without the need to rewrite these algorithms.

By default, Spark works standalone with local storage, but it also provides support for other cluster managers, such as Apache Mesos[23] and Hadoop YARN[35]. In addition, Spark contains specific scripts in order to enable the deployment of a standalone cluster within the Amazon EC2 Cloud environment. There also exist mechanisms to facilitate the development of Spark applications without the need to rewrite these algorithms.

The deployment of Spark in a distributed environment requires three types of components: Driver Program, Cluster Manager and Workers. The most common deployment requires one node to run the Driver Program component and another node to run the Cluster manager, while the rest of the pool to acts as Worker nodes.

C. Discussion

While COMPSs and Spark were both designed as programming models, they nevertheless have significant differences that are mainly due to different philosophies behind the design of the two environments.

1) Programming model: Both environments differ in the programmability conception: COMPSs offers a task-based programming model where each task represents a function defined by the user. On the other hand, Spark provides a set of operators ready to be applied to the RDDs, which have been internally optimized into a set of tasks.

Spark was designed with a set of powerful and optimized operators which developers have to use in order to develop any application. It possesses a large amount of functions that are internally designed to exploit the parallelism of each operation independently. Consequently, developers need to familiarize themselves with these functions and use them in all algorithms in order to get performance. This hampers the code reusability, since the algorithms already developed cannot be used in Spark due to their lack of calls to Spark operators, and the strict definition of functions forces developers to re-design the algorithm, with the consequent requirement for manpower.

A different design approach was adopted for COMPSs and was conditioned by code reusability and programmability. The aim of the COMPSs team was to provide an efficient programming model capable of finding inherent parallelism from the user code with a minimum amount of effort. With COMPSs, developers are not required to rewrite their current working code. An interface is offered for the definition of those functions that are becoming tasks, and all application and infrastructure management is delegated to the COMPSs runtime.

Moreover, Spark is regarded as RDD centric (all the operations are performed on the RDD partitions), whereas COMPSs considers the functions defined by the user as tasks. All operations over RDDs have the same defined inputs and outputs and require the implementation of the partition mechanism. Furthermore, COMPSs allows users to employ their own defined objects transparently without any partitioning requirement.

In addition to supporting the RDD operations, Spark provides operators for collective operations (e.g. for variable broadcasting). With this regard, COMPSs support for collective operations is limited to a task replication capability (if a task is defined with the “isReplicated” flag enabled, it will be executed in all nodes every time it is invoked).

With regard to the main differences, the three leading question that arise in this context are:

1) Will the existing Spark primitives be sufficient? The answer to this question is not easy to provide, since it depends on what Spark users wish to use.

2) What is the cost of migrating my existing application to Spark? The answer to this question is straightforward, because Spark forces users to use their own primitives, thereby making it necessary to rewrite large parts of the code. This code rewrite may even require some of the application parts to be redesigned, because of the data types or primitives. Alternatively, the migration of an application to COMPSs requires the identification of the functions to be used as tasks, and possible changes in its structure may also be necessary in order to improve application efficiency.

3) Which of the two can be considered more flexible? In the case of Spark, all operations over RDDs have the same defined inputs and outputs, whereas COMPSs is more flexible in this regard because it allows users to define as many parameters as necessary for the tasks (functions represented on the interface or decorated).

These questions are not so important when COMPSs is concerned because the user–defined functions are used as tasks. The only restriction being user imagination, thus obviating the need to rewrite the code, and using the user data structures. This enables the existing code to be parallelized in an easy and natural programming way. However, changes on the application structure may be needed in order to improve the application parallelization.

2) Architecture: The architectures are very similar conceptually, both frameworks define a main core runtime, where all the features are implemented. On top of them, we find bindings that provide support for other languages (e.g. Python in both frameworks, or C/C++ in COMPSs and Scala in Spark).

One of the salient advantages of Spark is the wide range of libraries that are available around it. Its wide popularity is
empowering the increase of these libraries due the growing demand of known algorithms of extensive use. In this sense, COMPSs constitutes a step backwards, since there is no existing library containing these functions, although it should be pointed out that COMPSs can use any existing function as a task, which enables the existing libraries and code to be reused.

Both of these frameworks include mechanisms for the interaction with storage technologies. In the case of Spark, it is ready to work with S3 and HDFS, while COMPS is ready to work with dataClay and Hecuba. Moreover, Spark also implements mechanisms to support Mesos and Yarn for cluster management rather than that built by default. Similarly, work on interoperability with Mesos is currently under development for COMPSs.

Regarding deployment, both Spark and COMPSs share a master-worker structure. However, two management nodes are recommended for Spark (one for the application driver and the other for the cluster management) due to performance reasons, thus increasing the nodes needed when compared with COMPSs. Furthermore, this capability of sharing the management across multiple nodes leads to scalability improvements. In this sense, the single master structure of COMPSs represents a barrier to scalability. This potential issue is expected to be attenuated by task nesting.

Communications within nodes or both frameworks are performed through point to point connections. In particular, the COMPSs communications for the requested data among tasks is done from master to worker, or among workers if possible (e.g. when the data requested by a task has been produced by another task in another worker). This communication among workers reduces the master data management pressure. In addition, the COMPSs scheduler tries to allocate tasks with data dependency on the same node in order to avoid any communication among workers and making the most of having the object in cache.

As regards the Python binding, both Spark and COMPSs invoke remote Python tasks from Java in the same way. This mechanism involves the creation of a process builder, which establishes a terminal and invokes the Python interpreter over a particular piece of code. When the execution has finished, the runtime captures the output and destroys the process. Performance analysis based on runtime trace files of COMPSs applications has been useful for determining that this mechanism is inefficient due to the new process creation and destruction. For this reason, COMPSs implements persistent workers for Java, and the team is working on introducing this improvement for Python as well.

IV. APPLICATIONS AND PERFORMANCE COMPARISON

We have studied the performance of Spark and COMPSs under the same conditions in terms of applications, amount of resources and hardware specifications. Our goal is to illustrate the performance that they both achieve, and then compare the results in order to analyse the advantages and disadvantages of both of when parallelizing the applications.

We first describe the infrastructure in which both frameworks are deployed, considering the applications, the experiments and the workloads. The performance results are also presented and commented. Finally, a brief summary of the comparison is provided.

A. Infrastructure

The infrastructure used in this comparison is the MareNostrum III supercomputer [11], located at the Barcelona Supercomputing Center (BSC). This supercomputer is composed of 3,056 nodes, each with two Intel SandyBridge-EP E5-2670/1600 20M (8 cores at 2.6 Ghz each), a main memory that varies from 32 to 128 GB, FDR-10 Infiniband and Gigabit ethernet network interconnections, and 3 PB of disk storage. It provides a service for researchers from a wide range of different areas, such as life sciences, earth sciences and engineering.

In this comparison, we requested up to 66 compute nodes with 32 GB of main memory and Infiniband network interconnection. More specifically, the configurations used in this comparison, taking into account the master-worker deployment architecture of both frameworks, are shown in Table I.

The COMPSs version used is 2.0 and SPARK4MN 3.10 (which internally uses Spark 1.5.2). Regarding Java, the version used is 1.8.0_66 64 bits.

<table>
<thead>
<tr>
<th># Framework</th>
<th># Master Nodes</th>
</tr>
</thead>
<tbody>
<tr>
<td>COMPSs</td>
<td>1 Master</td>
</tr>
<tr>
<td>SPARK</td>
<td>1 Master</td>
</tr>
<tr>
<td></td>
<td>1 Domain</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th># Worker Nodes</th>
<th># Cores</th>
<th>RAM (Mb)</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>16</td>
<td>32</td>
</tr>
<tr>
<td>2</td>
<td>32</td>
<td>64</td>
</tr>
<tr>
<td>4</td>
<td>64</td>
<td>128</td>
</tr>
<tr>
<td>8</td>
<td>128</td>
<td>256</td>
</tr>
<tr>
<td>16</td>
<td>256</td>
<td>512</td>
</tr>
<tr>
<td>32</td>
<td>512</td>
<td>1024</td>
</tr>
<tr>
<td>64</td>
<td>1024</td>
<td>2048</td>
</tr>
</tbody>
</table>

Table I: Infrastructure configurations
B. Applications

The applications considered for this comparison are Wordcount, Kmeans and Terasort, since they enable us to evaluate different workflows and requirements. Unfortunately, since COMPSs and Spark provide different programming models, the same application implementation cannot be considered among them. However, use has been made of similar application implementations that preserve their essential behaviour as far as possible, as well as the particular mechanisms of each programming model. The language chosen for this comparison is Java. However, Scala is considered on the Spark Kmeans and Terasort since they belong to well-known and efficiently developed Spark ecosystem libraries. Each of these applications is briefly described in the following subsections.

1) Wordcount: A simple algorithm like Wordcount is considered for the initial experimentation. This algorithm performs a word count across multiple files located within a folder (dataset) and stores the total amount of appearances of each word within a text file.

The Wordcount algorithm used for Spark is provided as an example in the documentation belonging to [10] (Algorithm 1). In greater detail, this algorithm performs one flatMap transformation that splits the text by spaces (line 2), and a map to pair transformation assigns the value of one to each of the words (line 8). Then, a reduce by key transformation is performed in order to accumulate all values of the same word (line 14). Finally, an action that saves the results as a text file (line 20) triggers the previous transformations.

On the other hand, the Wordcount algorithm used for COMPSs follows the schema of a sequential Wordcount code (Algorithm 2). In this implementation, two functions have been identified as tasks: Wordcount (which reads a file and returns a dictionary with the appearances of each word (line 18)) and reduce (which merges two dictionaries (line 35)). In this case, the first loop (line 4) performs one Wordcount per file, and then, the results are merged according to a reduce tree of pairs (line 8). Finally, the result of merging all dictionaries is saved as a file.

Both applications are developed in accordance with the way that any habitual developer would adopt for each framework, without any improvement in performance. This simple application enables us to show the programmatic differences that exist among frameworks (Algorithms 1 and 2). Both applications have the same input (a set of files) and also produce an output. This output is one file in the case of COMPSs, and since the RDDs partitioning is left to Spark, multiple output files are produced in its case.

Regarding the code size, the Wordcount implementation for Spark is coded in 46 lines, while 110 lines are required for COMPSs (plus 10 for the interface). The main reason for this greater number of lines in the COMPSs version is...
due to the fact that the user has to implement the classes and mechanisms for reading/writing the input/output datasets.

2) **Kmeans:** The next application considered is Kmeans, which is a well-known machine-learning algorithm (NP-hard), popularly employed for cluster analysis in data mining, and interesting for benchmarking and performance evaluation. The purpose of the Kmeans algorithm to group a set of multidimensional points into a predefined number of clusters, in which each point belongs to the closest cluster (with the nearest mean distance), in an iterative process.

In this case, we consider the Kmeans provided by the Spark machine-learning library MLlib [26]. The purpose of this library is to provide a set of machine-learning algorithms efficiently parallelized for Spark. Consequently, the Kmeans implementation takes the peculiarities of Spark into account for its performance enhancement.

In contrast, we have implemented a parallel version of Kmeans according to a similar approach, but following the COMPSs programming model. Briefly, our implementation parallelizes the algorithm taking into account that the entire dataset is divided into fragments (like Spark RDDs chunking). For each of the fragments calculates the distance of each point to all cluster centers in order to assign each point of the fragment to the nearest cluster. The dimensions of the points belonging to the same cluster are then added up and finally normalized. The result from this normalization provides the new centers, and they are compared with the previous iteration centers to check the convergence condition. All computation is performed distributively and the only object that is moved between the master and the workers on each iteration is the centers list.

Unlike the Wordcount algorithm, the Kmeans input dataset is produced during the execution of the application with random initialization with a specific seed. This enables us to check the behaviour of both frameworks when working exclusively with objects created during the execution without performing any access to the storage. Moreover, we measure the time required to perform 10 iterations for its evaluation on each execution.

The complexity of the Kmeans workflow is higher than the Wordcount workflow, with larger communications and computation.

Regarding the code size, the Kmeans implementation for Spark is coded in 871 lines (considering the application and the necessary code from MLlib). On the other hand, Kmeans for COMPSs is coded in 503 lines (plus 35 for the interface). Since the parallelization of Kmeans is not trivial, the use of operators defined by Spark forces a complex structure and algorithm, which require more lines of code than a sequential version. In this application, 7 transformations and 4 actions were required in the Spark implementation. In COMPSs, only 4 functions were taskified.

3) **Terasort:** The last application considered in this evaluation is Terasort. This application sorts a large input dataset.

In particular, the Terasort used for this evaluation [12] is fed by an input dataset generated by the Teragen tool (available within the Terasort package), which produces multiple binary files containing multiple 100 bytes key-value entries (10 bytes key and 90 bytes value). In the first step, Terasort performs a data partitioning of the input dataset by the entry key, and then uses the sortByKey transformation provided by Spark. To trigger the execution, saves the output result to disk.

The Terasort implementation for COMPSs follows a similar approach. It performs the partitioning of the data into a set of buckets depending on the entry key, and then sorts each bucket. The implementation consists of four tasks: one that filters an input file and retrieves the entries within a given range (bucket); one that reduces the filtering tasks (to know when a bucket is complete); another that sorts an entire bucket; and finally a task that checks that all buckets have been sorted. Since the current COMPSs implementation does not properly handle dependences from data in collections, a shared disk across the worker nodes is required for storing partial sorting results within buckets folders. This requirement is not present in Spark implementation, since its runtime is able to perform shuffling automatically.

In Spark, the Terasort is designed to use the same input dataset generated by Teragen in order to provide a fair comparison. The Terasort input and output are files, as opposite to the Kmeans, and with higher requirements for storing the results than Wordcount.

Regarding the code size, Spark implementation is performed in 259 lines of code, while the COMPSs implementation requires 508 (plus 34 for the interface). This lower requirement of lines in the Spark implementation is due to the fact that it already provides a transformation for this purpose (sortByKey), which is used after partitioning the input dataset (with a partitionBy transformation) and is triggered by the result saving. As in the Wordcount example, the COMPSs implementation has to deal directly with files reading/writing, which increases the amount of code lines.

C. Experiments and datasets

The Wordcount, Kmeans and Terasort applications are executed within the infrastructure described in Section IV-A, with different configurations to perform a more thorough analysis when evaluating their performance and scalability.

For each application, we have considered two main experiments for strong and weak scalability. For the strong scalability experiments the dataset is kept fixed, while the amount of worker nodes is increased. On the other hand, the input dataset is increased in the same proportion as the amount of worker nodes for the weak scalability experiments. The larger dataset used for the maximum amount of worker nodes in the weak scaling experiments matches the dataset used for the strong scalability experiments.
Table II: Wordcount dataset configurations

<table>
<thead>
<tr>
<th># Worker nodes</th>
<th># Files (Strong)</th>
<th># Files (Weak)</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>1024</td>
<td>16</td>
</tr>
<tr>
<td>2</td>
<td>1024</td>
<td>32</td>
</tr>
<tr>
<td>4</td>
<td>1024</td>
<td>64</td>
</tr>
<tr>
<td>8</td>
<td>1024</td>
<td>128</td>
</tr>
<tr>
<td>16</td>
<td>1024</td>
<td>256</td>
</tr>
<tr>
<td>32</td>
<td>1024</td>
<td>512</td>
</tr>
<tr>
<td>64</td>
<td>1024</td>
<td>1024</td>
</tr>
</tbody>
</table>

Table III: Kmeans configuration parameters

<table>
<thead>
<tr>
<th># Worker Nodes</th>
<th># Points (M) (Strong)</th>
<th># Fragments</th>
<th># Points (M) (Weak)</th>
<th># Fragments</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>131</td>
<td>1024</td>
<td>2</td>
<td>16</td>
</tr>
<tr>
<td>2</td>
<td>131</td>
<td>1024</td>
<td>4</td>
<td>32</td>
</tr>
<tr>
<td>4</td>
<td>131</td>
<td>1024</td>
<td>8</td>
<td>64</td>
</tr>
<tr>
<td>8</td>
<td>131</td>
<td>1024</td>
<td>16</td>
<td>128</td>
</tr>
<tr>
<td>16</td>
<td>131</td>
<td>1024</td>
<td>32</td>
<td>256</td>
</tr>
<tr>
<td>32</td>
<td>131</td>
<td>1024</td>
<td>65</td>
<td>512</td>
</tr>
<tr>
<td>64</td>
<td>131</td>
<td>1024</td>
<td>131</td>
<td>1024</td>
</tr>
</tbody>
</table>

Table IV: Terasort configuration parameters

<table>
<thead>
<tr>
<th># Worker nodes</th>
<th># Files (Strong)</th>
<th># Files (Weak)</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>256</td>
<td>4</td>
</tr>
<tr>
<td>2</td>
<td>256</td>
<td>8</td>
</tr>
<tr>
<td>4</td>
<td>256</td>
<td>16</td>
</tr>
<tr>
<td>8</td>
<td>256</td>
<td>32</td>
</tr>
<tr>
<td>16</td>
<td>256</td>
<td>64</td>
</tr>
<tr>
<td>32</td>
<td>256</td>
<td>128</td>
</tr>
<tr>
<td>64</td>
<td>256</td>
<td>256</td>
</tr>
</tbody>
</table>

For the Wordcount application, the dataset considered for the strong scalability experiments is composed of 1024 plain text files of 1 GB each, which represent 1 TB of input dataset. For the weak scalability experiments it goes from 16 GB to 1 TB (the same amount of files as cores available on each worker node) (Table II). Each file is generated using the *loremipsum* Python library.

For the Kmeans application, the dataset considered for the strong scalability experiments is composed of 131,072,000 points of 100 dimensions, 1,000 centers and 1,024 fragments. Since the size of each point dimension is 8 bytes (double), the total amount of the dataset in memory is \( \sim 100 \) GB plus the data structures. Alternatively, the dimensions and centers are kept constant, while the points and fragments vary depending on the amount of worker nodes available for each weak experiment (Table III).

Since we are interested in evaluating the performance of the frameworks, the Kmeans executions are limited to 10 iterations instead of executing the application until the convergence condition is reached. Furthermore, the dataset initialization is set to random with a seed.

Finally, the Terasort is evaluated with an input dataset composed of 256 files of 1 GB each for the strong scaling experiments. They are generated by the Teragen Spark application. In this scenario, it is left to Spark to decide the amount of fragments that obtain the best performance (which depends on the RDDs partitioning). Since COMPSs does not perform any automatic partitioning of the data, the number of fragments and buckets need to be defined explicitly. In this case, we have considered 4 fragments and 256 buckets. Alternatively, the amount of files is varied according to the amount of worker nodes for the weak scaling experiments (Table IV).

Both the input and output of the executions are from and to the General Parallel File System (GPFS) shared disk in MareNostrum.

Each experiment and configuration has been evaluated 10 times with all loggers disabled.

D. Results

The applications are executed in the described infrastructure for each of the configurations. In the following subsections, the results of the experiments are thoroughly analysed.

1) Performance: The legitimate objective of both frameworks is to parallelize the applications in order to improve performance. In this sense, we present the performance results obtained through violin plots with the results from Spark and COMPSs overlayed. A violin plot is a boxplot combined with a histogram per experiment kernel. The information provided by the histograms enable users to see the duration probability with each amount of worker cores in COMPSs and Spark for the analysed application and experiment. Internally, they include a thin vertical line that represents the 95% confidence interval, a thick vertical line that represents the interquartile range, and a white dot for the median.

A much better performance is observed with COMPSs in the Wordcount application with all defined configurations on both strong and weak experiments (Figure 5). The way that both frameworks process the application workflow seems to be the main reason for this performance difference. However, the lower Spark performance may be due to the way in which Spark performs the RDD partitioning in this particular case. Spark reads all files and performs its partitioning, while in COMPSs each file represents a chunk and is read within the task. This partitioning gives rise to a huge amount of tasks in the Spark Wordcount application, thus increasing their management and communication needs. Moreover, as a result of this partitioning schema, Spark
creates a large amount of output files, most of which are empty or contain only one word and its appearances. The creation of such an amount of files hampers the achievement of better performance, since it requires a large amount of access to disk. The only way to prevent this behaviour is to tune the application and include specific information of the input dataset and partitioning, which would make the application efficient for each specific dataset, but limited to that dataset. Moreover, this requirement means that users must know details of Spark internals in order to achieve good performance results, which makes it rather unfriendly for new developers, since they need to do more than simply adapt to their operators.

COMPSs is able to achieve a better performance, given the Wordcount that a sequential developer would program. The COMPSs approach does not hide the partitioning from users, who can deal with it by processing the entire file per Wordcount or parts of each if necessary. This is an important consideration, since the application has not suffered any modification and only an interface is required.

We would like to point out that Spark is unable to execute the Wordcount with the largest dataset with only one worker node, thus raising a memory error exception.

This large performance difference is not been observed with the Kmeans application. In this application, the dataset is generated in memory during the execution, thereby leveraging the amount of access to disk. In the strong scaling experiment, we are unable to execute the application with fewer than 256 worker cores (16 worker nodes) for any of the frameworks. While Spark is been able to achieve better than average performance with 256 worker cores, COMPSs overtakes Spark slightly when we increase the amount of worker nodes.

With regard to the weak scaling experiment, the results show that COMPSs is able to achieve a better performance in all cases.

For the Terasort example in the strong scaling experiment, we have only been able to execute it successfully with Spark with the largest amount of worker cores (1,024) due to memory issues. A similar issue appears in COMPSs with fewer than 128 worker cores (8 worker nodes). For this experiment, and with 1,024 worker cores (64 worker nodes), COMPSs outperforms Spark.

In the weak scaling experiment, the results are uneven for both frameworks. Spark is able to achieve better performance with fewer than 512 worker cores, but COMPSs outperforms Spark from 256 worker cores. Additionally, the time spent by Spark increases exponentially for the largest deployment configurations (512/32 and 1,024/64 worker cores/nodes), while for COMPSs an increase in time is even observed and follows a linear behaviour.

2) Variability: With regard to variability (Figure 5), the results show that Spark presents the highest variability with the Wordcount application, which is not present when using COMPSs. However, this variability is lower in Kmeans and Terasort.

Although the Wordcount and Terasort applications require access to input and output datasets, the Terasort applications is enhanced for reading and writing binary data, which may lead to a variability reduction from the I/O reading and writing operations.

In general, both frameworks present a low variability.

3) Scalability: Speed-up and efficiency are calculated for the strong and weak experiments (see Figures 6, 7 and 8). While the speed-up is computed according to the results obtained with Spark, the efficiency is computed for each framework with regard to itself with one worker node.

The speed-up plots show better results for COMPSs in the Wordcount and Kmeans examples. It is interesting to observe that in the Wordcount application, Spark shows a flat curve in the speed-up for the strong scaling experiment, and a decreasing one for the weak scaling experiment. The situation improves a little for the Kmeans application in Spark, but still with better behavior than that shown by COMPSs. In the weak scaling experiments, COMPSs achieves slightly better speed-up in both cases, although the overall behaviour is quite similar and the performance decays in almost the same proportion when the worker node counts are increased.

For the Terasort application, we only show the speed-up and efficiency curves for the weak scaling experiment, since for the strong scaling case we are only able to measure the performance with 64 nodes. The curves show the uneven behavior already commented on in previous paragraphs.

V. CONCLUSION

Enabling users to tackle Big Data problems is a challenging issue for which Spark is currently regarded as the Goliath framework due to its wide popularity and adoption. As an alternative, COMPSs is the contender as the David competing for the throne of Big Data programming models. To this end, COMPSs provides support for exploiting the inherent parallelism of existing sequential applications execution in distributed environments, as opposed to Spark, which forces users to use their pre-defined operators. This means that the existing applications must be rewritten and the existing algorithms adapted to the operators, which in some cases can be difficult.

In the experiments performed, COMPSs has proven capable of achieving better performance and scalability in most cases when compared with Spark. In applications like Wordcount, the difference is indeed significant. On the
other hand, the Kmeans and Terasort applications achieve a similar performance, and although Spark has shown a better performance for small datasets with a reduced set of worker nodes, COMPSs is able to overtake it for larger datasets and greater numbers of worker nodes.

The details of the performance comparison becomes difficult to analyse due to the lack of tools for analysing Spark behaviour and the reduced amount of metrics that...
it provides. In this regard, COMPSs is instrumented with Extrae, which is able to generate runtime trace files with a rich amount of performance metrics. This is not only interesting for performance comparison, but also important for analysing the application behaviour and bottleneck detection.

Although the current results show that COMPSs is able to achieve better performance and scalability than Spark, we are convinced that we can improve the COMPSs results presented in this work by enhancing the scheduling policy. Nevertheless, one of the main advantages of Spark is the rich ecosystem that it features. In this ecosystem, the growing amount of libraries leverages users from implementing the most common applications.

In a nutshell, the main advantages of COMPSs are as follows: it enables a natural development without operators, keeps the sequential functioning of the applications and is able to achieve a better performance than Spark. Alternatively, the Spark advantages are: it enjoys a wide popularity and its adoption strengthens the growth of its ecosystem. On the other hand, the fact that COMPSs requires users to define the tasks may constitute its main drawback. This flexibility, which empowers programmability, has a direct performance impact that relies on a good selection of the functions to be used as tasks and their parameters. However, improvement on this may be gained by the fact that COMPSs is able to generate the task dependency graph resulting from any application execution and runtime trace files with Extrae, which allows users to analyse the application behaviour with the Paraver tool. Spark’s main disadvantage is the need to use its pre-defined operators, which makes code migration difficult because it requires the rewriting of a new code that only works with Spark.

Future work will consider the evaluation of these two frameworks with Python applications; that is, through the existing python bindings (PyCOMPSs vs. PySPARK) and other distributed frameworks different from SPARK. Another future work line could be to extend the comparison with other applications.

In addition, we have detected inefficiencies in the execution of reduce phases by the COMPSs runtime and plan to implement strategies to improve this situation by providing a syntax for reductions similar to the user-defined reductions in OpenMP, as well as implementing an efficient mechanism in the runtime to handle them.
This work is supported by the Spanish Government (SEV2015-0493), by the Spanish Ministry of Science and Innovation (contract TIN2015-65316-P), by Generalitat de Catalunya (contracts 2014-SGR-1051 and 2014-SGR-1272). Javier Conejero’s postdoctoral contract is cofinanced by the Ministry of Economy and Competitiveness under the Juan de la Cierva Formación postdoctoral fellowship number FJCI-2013-11140. This work is also supported by the Intel-BSC Exascale Lab. The Human Brain Project receives funding from the EU’s Seventh Framework Programme (FP7/2007-2013) under grant agreement no 604102.

REFERENCES


[7] JNI APIs and Developer Guides. Web page at https://docs.oracle.com/javase/8/docs/technotes/guides/jni/, (Date of last access: 19th December, 2016).


AUTHORS’ BIO

Javier Conejero is a Senior Researcher at Barcelona Supercomputing Center. He holds a Ph.D. Degree in Computer Science from the University of Castilla-La Mancha. Javier worked at CERN for one year on WLCG software development and management. His current research interests are programming models, efficient exploitation of Cloud and HPC environments and QoS.

Sandra Corella is a Junior Developer at Barcelona Supercomputing Center. She received her B.E. in Computer Science from the Polytechnic University of Catalonia and is currently pursuing her M.E. in Advanced Mathematics and Mathematical Engineering. Her current research interests are programming models, parallel programming and scientific algorithms.

Rosa M Badia is the manager of the Workflows and Distributed Computing research group at the Barcelona Supercomputing Center (BSC). She is also a Scientific Researcher from the Consejo Superior de Investigaciones Científicas (CSIC). She holds a PhD on Computer Science (1994) from the Technical University of Catalonia (UPC). Her current research interests are programming models for heterogeneous systems and distributed computing, and convergence between HPC systems and Big Data.

Jesus Labarta has been a full professor of Computer Architecture at UPC since 1990. Since 2005, he has been responsible for the Computer Science Research Department within BSC. His major directions of current work relate to performance analysis tools, programming models and resource management. His team distributes the Open Source BSC tools (Paraver and Dimemas) and conduct research on increasing the intelligence embedded in the performance analysis tools. He is involved in the development of the OmpSs programming model and its different implementations for SMP, GPUs and cluster platforms and in the COMPSs programming model for distributed computing.


