Contents lists available at ScienceDirect

Future Generation Computer Systems

journal homepage: www.elsevier.com/locate/fgcs

Efficient development of high performance data analytics in Python

Javier Álvarez Cid-Fuentes [a,*], Pol Álvarez [a], Ramon Amela [a], Kuninori Ishii [b], Rafael K. Morizawa [b], Rosa M. Badia [a,c]

[a] *Barcelona Supercomputing Center (BSC), Spain*
[b] *Fujitsu, Ltd., Japan*
[c] *Artificial Intelligence Research Institute (IIIA), Spanish National Research Council (CSIC), Spain*

ARTICLE INFO

Article history:
Received 17 October 2018
Received in revised form 27 August 2019
Accepted 27 September 2019
Available online xxxx

ABSTRACT

Our society is generating an increasing amount of data at an unprecedented scale, variety, and speed. This also applies to numerous research areas, such as genomics, high energy physics, and astronomy, for which large-scale data processing has become crucial. However, there is still a gap between the traditional scientific computing ecosystem and big data analytics tools and frameworks. On the one hand, high performance computing (HPC) programming models lack productivity, and do not provide means for processing large amounts of data in a simple manner. On the other hand, existing big data processing tools have performance issues in HPC environments, and are not general-purpose. In this paper, we propose and evaluate PyCOMPSs, a task-based programming model for Python, as an excellent solution for distributed big data processing in HPC infrastructures. Among other useful features, PyCOMPSs offers a highly productive general-purpose programming model, is infrastructure-agnostic, and provides transparent data management with support for distributed storage systems. We show how two machine learning algorithms (Cascade SVM and K-means) can be developed with PyCOMPSs, and evaluate PyCOMPSs' productivity based on these algorithms. Additionally, we evaluate PyCOMPSs performance on an HPC cluster using up to 1,536 cores and 320 million input vectors. Our results show that PyCOMPSs achieves similar performance and scalability to MPI in HPC infrastructures, while providing a much more productive interface that allows the easy development of data analytics algorithms.

1. Introduction

The amount of data that society as a whole generates is growing extremely fast. It is expected that by 2020, the amount of useful data will be of more than 16 zettabytes (i.e., $16 \cdot 10^{12}$ GB) [1]. Extracting useful information from this *big data* can have an enormous impact on many societal activities, such as healthcare [2], manufacturing [3], or city planning [4]. In addition to this, analyzing large amounts of data is becoming crucial in numerous research areas, such as biology [5], astronomy [6], or high energy physics [7] among many others [8,9].

The process of extracting useful information from large amounts of data is also known as *big data analytics* (BDA) [10]. BDA involves transforming the data using various operations, such as sorting, aggregating, or filtering [11]; as well as using machine learning algorithms to obtain new information and to discover patterns in data [12,13]. Key challenges in BDA include representing heterogeneous data efficiently, eliminating data redundancies, storing large amounts of data in an scalable and fault-tolerant manner, building scalable algorithms, transmitting large quantities of data efficiently, and visualizing high dimensional data [9,14]. These challenges have motivated the appearance of several BDA especialized tools [10,15], such as data oriented programming models, like MapReduce [16] and Apache Spark [11]; NoSQL databases like Apache Cassandra [17]; and distributed file systems such as the Hadoop file system [18].

Although BDA has become crucial in many scientific fields, traditional scientific computing tools do not provide means for efficient BDA. In addition to this, existing BDA tools are not designed for traditional scientific computing facilities, such as *high performance computing* (HPC) clusters [10]. This gap between BDA and HPC is especially severe in programming models [10]. On the one hand, scientific computing programming models such as OpenMP [19] and Message Passing Interface (MPI) implementations [20] are not oriented for data management, and lack the productivity required for fast development of BDA algorithms. On the other hand, existing BDA frameworks [11,16] are not compatible with some HPC components, like batch queue systems, and thus require increased integration efforts to be deployed in HPC clusters [21]. This creates the need for a general-purpose

* Corresponding author.
E-mail address: javier.alvarez@bsc.es (J. Álvarez Cid-Fuentes).

https://doi.org/10.1016/j.future.2019.09.051
0167-739X/© 2019 Elsevier B.V. All rights reserved.

programming model to write and execute BDA algorithms in HPC infrastructures with minimal developer effort.

In this paper, we propose and evaluate PyCOMPSs [22] as an approach for bridging the gap between BDA and HPC programming models. PyCOMPSs is a task-based programming model that can be used to easily build and execute parallel Python applications. On the one hand, PyCOMPSs offers a more productive API than traditional scientific computing programming models like MPI and OpenMP. On the other hand, unlike existing BDA tools like Spark, PyCOMPSs is completely compatible with HPC infrastructures. Moreover, PyCOMPSs allows developers to write new distributed machine learning or BDA algorithms from scratch, as well as to parallelize their existing Python codes. This means that PyCOMPSs is not tied to a specific class of algorithms, and has a much lower risk of becoming obsolete than current distributed machine learning libraries. Finally, PyCOMPSs has the advantage of being based on Python, which provides great productivity, and is one of the most popular programming languages among data scientists [23].

The main contributions of this paper are: (i) we demonstrate how BDA algorithms can be easily developed in PyCOMPSs; (ii) we show how these algorithms can be deployed and executed in an HPC scenario with PyCOMPSs; and (iii) we evaluate the productivity and performance of PyCOMPSs for BDA in HPC infrastructures. As part of this evaluation, we compare PyCOMPSs with MPI.

In Section 2, we summarize the related work, highlight the limitations of existing approaches for BDA in HPC infrastructures, and explain how PyCOMPSs overcomes these limitations. In Section 3, we describe PyCOMPSs programming model, and discuss the features that make PyCOMPSs ideal for BDA in HPC infrastructures. In Section 4, we present the PyCOMPSs implementation of two well-known machine learning algorithms: Cascade SVM and K-means. In Section 5, we compare the code complexity of these implementations with the same algorithms in MPI. In Section 6, we compare the performance of PyCOMPSs and MPI, and in Section 7 we present our conclusions.

2. Related work

A widely used distributed programming paradigm for HPC clusters is the Message Passing Interface (MPI) [20]. Although MPI can be used to build machine learning and data analytics algorithms [24–26], it requires developers to manually handle communications, data transfers, and load balancing. In contrast, PyCOMPSs automatically manages all these aspects in a transparent way, and offers a much simpler programming model that abstracts the developer from the parallelization details.

The most prominent framework for BDA is Apache Spark [11]. Spark is based on the idea of *resilient distributed datasets* (RDD), which are distributed data structures that can be operated in parallel. Spark provides a productive programming model that can be used to write many BDA algorithms using a relatively small set of operations on RDDs, such as *map*, *reduce*, and *filter*. Additionally, Spark keeps data in memory when possible for speeding up accesses. The disadvantages of Spark are that it is difficult to deploy on HPC clusters [21], shows poor performance in certain scenarios [27,28], and enforces a specific program abstraction on developers.

A recent framework related to Spark is PiCo [29]. PiCo unifies batch and stream data access models, and defines programs as sequences of parallel operations over data. Like in Spark, these operations are predefined in PiCo's API. Compared to Spark, PiCo provides analogous productivity by offering a similar set of operations, while achieving performance improvements in HPC infrastructures, both in terms of execution time and memory footprint.

Contrary to Spark and PiCo, PyCOMPSs does not enforce a specific program abstraction or API on developers. With PyCOMPSs, developers can build parallel programs with an arbitrary structure using only a small set of annotations. PyCOMPSs has more expressive power than Spark and PiCo, and does not require users to re-think their problem to tailor it to a specific structure.

The High Performance Analytics Toolkit (HPAT) [27] is a BDA framework specifically designed for HPC infrastructures. HPAT is based on the parallelization of vector operations, which are common in many machine learning algorithms. HPAT provides an API for data access, and then is able to automatically parallelize matrix–vector and matrix–matrix operations carried out on loaded data. More precisely, HPAT generates MPI/C++ code from high level algorithm definitions in Julia and Python. In this way, HPAT provides high productivity and MPI/C++ comparable performance on HPC clusters without enforcing a specific API on developers like Spark and PiCo.

HPAT is more similar to PyCOMPSs in the sense that it automatically detects the parallelism of applications through code annotations. However, HPAT only detects parallelism in vector operations, while PyCOMPSs can be used to parallelize any algorithm that can be expressed as a collection of tasks.

Swift/T [30] is a programming language and distributed runtime that allows the execution of parallel workflows on distributed platforms, including HPC clusters. Swift/T's distributed design provides great scalability, both in number of parallel tasks and computational resources. Swift/T applications are written in Swift language, and the distributed execution is driven by MPI. PyCOMPSs is different to Swift/T in that it does not require developers to learn a new programming language. Instead, users just need to insert simple code annotations on their regular Python applications.

Other existing BDA frameworks, like JS4Cloud [31], are especially designed for cloud infrastructures, and their integration with HPC clusters is unclear.

Compared to existing approaches, PyCOMPSs provides the best trade-off between flexibility, productivity, and performance. PyCOMPSs offers a highly productive programming model that does not force developers to accommodate their problem to a specific API, or to learn a new programming language. Additionally, PyCOMPSs supports distributed storage systems, and can execute in multiple platforms and architectures.

3. PyCOMPSs overview

PyCOMPSs [22] is a task-based programming model that makes the development of parallel and distributed Python applications easier. PyCOMPSs consists of two main parts: programming model and runtime. The programming model provides a series of simple annotations that developers can use to define potential parallelism in their applications. The runtime analyzes these annotations at execution time, and distributes the computation automatically among the available resources. The main component of PyCOMPSs' programming model is the *task* annotation, which defines units of computation that can be executed remotely. Since PyCOMPSs' runtime is written in Java, Python syntax is provided through a binding.

3.1. Programming model

PyCOMPSs applications are regular Python applications with certain annotations that help the runtime to exploit parallelism. PyCOMPSs applications consist of two parts: main sequential code and task definitions. The main sequential code is the entry point of the application, while task definitions are just annotated functions. To mark a function as a task in Python, we employ the

```
def main:
    s = 0

    for i in range(10):
        s += multiply(i, i)

@task(num1=IN, num2=IN, returns=int)
def multiply(num1, num2):
    return num1 * num2
```

Fig. 1. Example PyCOMPSs application.

`@task` decorator as shown in Fig. 1. The `@task` decorator can be applied to any kind of function, including class methods, and annotated functions can be used as regular functions in the main code of the application.

The `@task` annotation can take various arguments. The only mandatory arguments are the *direction* of the task parameters and the type of the returned value. Task parameters can be primitive types, such as integers and floats, files, and serializable objects. This includes objects from widely used libraries such as NumPy and scikit-learn [32]. The direction argument defines if a parameter is read-only, write-only, or both, and thus can take three values for regular Python objects: `IN`, `OUT`, and `INOUT`; and three values for files: `FILE`, `FILE_OUT`, or `FILE_INOUT`. If not specified, the parameter is assumed to be `IN`. PyCOMPSs also supports the use of `*args` and `**kwargs` as input parameters for tasks.

PyCOMPSs' programming model also provides the `@openmp`, `@mpi`, and `@binary` decorators. These can be used to create tasks that run different types of binaries. In addition to this, hardware and software requirements for tasks can be defined using the `@constraint` decorator. In this manner, tasks can be forced to be scheduled in a particular type of resource, such as a GPU. Finally, PyCOMPSs also provides a minimal API to insert synchronization points. This can be done with a function called `compss_wait_on`.

The simplicity of PyCOMPSs' programming model allows for fast development of data analytics algorithms in a highly productive language that is widely used in the scientific community, and that is surrounded by a large ecosystem of mathematical libraries [32,33]. Moreover, any existing Python application can be easily parallelized by just including some annotations in the code.

3.2. *Runtime*

PyCOMPSs' runtime follows a master–worker approach. The master process executes the main code of the application, and distributes computational work to a series of remote workers. Fig. 2 presents a diagram of the execution of a PyCOMPSs application.

The master process intercepts calls to annotated functions and inserts tasks in a data dependency graph instead of executing the function code. The master infers data dependencies from the direction of the task parameters, where write-after-write and write-after-read dependencies are avoided using renaming. Inserting tasks into the dependency graph is an asynchronous process, that is, objects returned by tasks are treated as future objects in the main code of the application. The master process can retrieve the actual value of task results by calling `compss_wait_on(object)`. This call waits for task completion and retrieves the object from the remote node.

In parallel with the task generation process, the master schedules tasks as they become dependency-free in the dependency graph. By default, PyCOMPSs uses a first-in-first-out scheduling policy that maximizes data locality. After scheduling a task, the master ensures that necessary data is transferred to the worker node. Transfers can happen between the master and the workers, and between different workers. Objects are serialized and written to disk to transfer them between different memory spaces. Apart from this, PyCOMPSs can be configured to use distributed file systems or distributed storage systems like Redis [34] and dataClay [35]. In the case of distributed file systems, the master assumes that workers have access to all files, and does not transfer them. In the case of distributed storage systems, PyCOMPSs supports storing objects in memory to reduce the overhead of disk accesses. This can speedup data analytics applications that apply sequences of transformations to multiple data in parallel.

PyCOMPSs' runtime provides fault-tolerance through task resubmission and rescheduling. In addition to this, PyCOMPSs offers live monitoring, and supports generating post-mortem execution Paraver [36] traces.

Finally, PyCOMPSs is infrastructure-agnostic, that is, PyCOMPSs applications can run in different infrastructures without source code modifications. This includes clouds, clusters, grids, and containerized platforms. To achieve this, PyCOMPSs supports communication with numerous resource managers, from Slurm [37] to Apache Mesos [38]. In addition to this, PyCOMPSs supports heterogeneous architectures, including GPUs and FPGAs [39].

PyCOMPSs' design makes it an excellent solution for BDA in HPC clusters. On the one hand, PyCOMPSs' dynamic task scheduling maximizes resource utilization in the presence of load imbalance. This is relevant for data parallel applications where processing time depends on the nature of the data. Other execution frameworks that allocate work statically, like MPI, achieve less resource usage in these scenarios. On the other hand, PyCOMPSs serializes data to disk unless a distributed storage system is used. This can be less efficient than keeping data in memory, but allows PyCOMPSs to handle much larger datasets than other *memory-oriented* frameworks like Spark.

4. Data analytics with PyCOMPSs

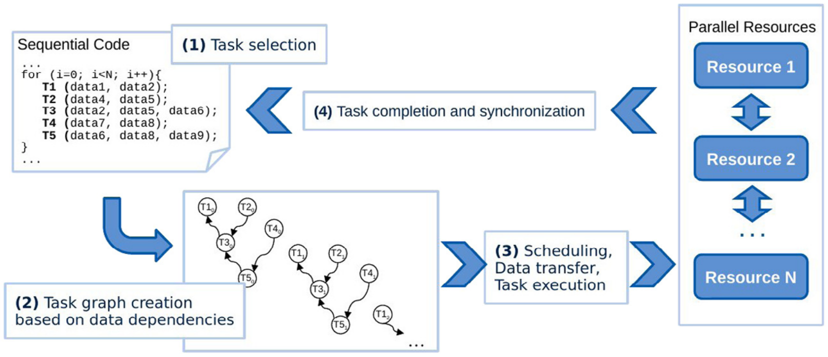
In this section, we show how data analytics algorithms can be easily developed with PyCOMPSs. More precisely, we implement two well-known machine learning algorithms: K-means [40] and Cascade Support Vector Machines (C-SVM) [41]. K-means is a clustering algorithm, while C-SVM is a parallel version of the support vector machine algorithm for classification [42]. For each algorithm, we first give a brief description, and then explain their implementation.[1]

4.1. *K-means*

The goal of clustering algorithms is to partition a set of n input vectors into groups or *clusters*, maximizing the similarity of the vectors of the same group with respect to other groups. Different similarity (or distance) functions end up producing a wide variety of clustering models. K-means is an iterative centroid model. This means that clusters are represented by a single vector, which is called the cluster center. The idea behind K-means is to start with a predefined number of random centers, and then refine them in a series of iterations. In the most general case, K-means employs the squared Euclidean distance as similarity measure.

Each iteration of the algorithm consists of two steps: assignment and update. In the assignment step, every vector is assigned to its nearest center. This assignation defines the vector *label*. In

[1] Source codes with sample data available at https://github.com/bsc-wdc/pycompss_bda.



```
...
for (i=0; i<N; i++){
    T1 (data1, data2);
    T2 (data4, data5);
    T3 (data2, data5, data6);
    T4 (data7, data8);
    T5 (data6, data8, data9);
}
...
```

Sequential Code
(1) Task selection
(4) Task completion and synchronization
Parallel Resources
Resource 1
Resource 2
Resource N
(2) Task graph creation based on data dependencies
(3) Scheduling, Data transfer, Task execution
T1
T2
T3
T4
T5

Fig. 2. PyCOMPSs task life-cycle.

the update step, the set of centers is refined via some kind of operation. In the most common case, new centers are obtained by computing the mean of the elements with the same label. These two steps alternate for a fixed number of iterations or until convergence (i.e., when the update step does not produce a significant change in the set of centers).

The K-means algorithm has two main drawbacks: first, the algorithm may stop at local optimums, and thus it highly relies on the random initialization of the centers; second, the number of clusters needs to be defined *a priori*, but it is unknown in many cases. Despite this, K-means remains a very popular clustering algorithm for various reasons. On the one hand, K-means results are stable enough for analyzing datasets and to summarize their main characteristics. On the other hand, clustering is an NP-hard problem and K-means is a highly efficient algorithm, with $\theta(m \cdot n)$ complexity, where m is the amount of centers and n the amount of input vectors.

The K-means implementation in PyCOMPSs consists of three tasks: `generate_fragment`, `cluster_points_sum`, and `merge_reduce_task`. The `generate_fragment` task generates a random set of vectors, and is used to generate the input dataset in a distributed manner. Alternatively, the K-means application also supports generating a random dataset in the main code of the application, and then distributing partitions among the workers. The `generate_fragment` task could be easily modified to read the input dataset from a file. Nevertheless, using a random dataset does not affect the performance of the algorithm.

The code for the `cluster_points_sum` task can be seen in Fig. 3. This task gets a list of vectors (`fragment_ponts`) and a list of centers, computes the nearest center of every vector (line 5), and then adds up and counts the vectors that *belong* to each center (line 8). This step is parallelized by dividing the input dataset into multiple partitions, and running one `cluster_points_sum` task per partition. Each of these tasks return a Python dictionary (i.e., a key–value structure) with center identifiers as keys, and a tuple of the number of vectors and the sum of these vectors as values. The algorithm then performs a parallel reduction using the `merge_reduce_task` task. In this way, the amount of parallelism is maximized as the computation of the new centers can start as soon as the distance computations of each partition finish.

Fig. 4 shows the code of the `merge_reduce_task` task. This task takes a variable number of input dictionaries (returned by `cluster_points_sum` tasks), and merges them into a new dictionary. The algorithm performs the merging process in parallel by creating an inverted tree of `merge_reduce_task` tasks until a single dictionary remains.

The main code of the K-means application is presented in Fig. 5. Variable `X` represents the input dataset. In the case of generating the input dataset in the master process, `X` contains a list of the partitions. In the case of generating the input data in a distributed manner using a number of `generate_fragment` tasks, `X` contains a list of future objects that represent these partitions. The application starts by generating random centers (line 3). Then, the application iterates until a fixed number of iterations or until convergence (line 7). In each iteration, the application computes the partial sums of each partition given the current centers (line 11–12). To achieve this, the application spawns a `cluster_points_sum` task for each partition (line 12). Then, the application merges together the result of the `cluster_points_sum` tasks in a reduction process (line 14). The `merge_reduce` function iteratively calls the `merge_reduce_task` task until a single dictionary of centers, number of vectors, and sum of these vectors is left. Finally, the application divides the sum of the vectors by the number of vectors to get the new mean of each center (line 16).

Note that the sum of vectors of each partition is processed in parallel, and that the reduction process accumulates these sums also in a distributed manner. Synchronization only occurs at the end of the iteration to get the final sum of vectors (line 15). In this manner, the application never transfers vectors, but means and center identifiers. This helps to reduce communication costs, especially in large datasets.

4.2. Cascade support vector machines

As said before, C-SVM is a parallel version of support vector machines (SVM), a widely used classification algorithm. The objective of a classification algorithm is to build a decision function from a set of input vectors that belong to a specific category. Then, this decision function can be used to categorize other vectors whose category is unknown. The process of building the decision function is called *training*, and the process of categorizing unknown vectors is called *prediction*. Different categories are also known as *labels*, and vector dimensions are called *features*. In the case of SVM, the decision function is a function of a subset of the input vectors, which are called the *support vectors*. The SVM algorithm finds the subset of the input vectors that better represent the categories in the input dataset, which are typically two. Finding the support vectors of a given dataset is a quadratic optimization problem.

The main idea behind the C-SVM algorithm is to split the set of input vectors into N partitions, and then find the support vectors of each partition in parallel to obtain N sets of support vectors. These N sets of support vectors are then combined together in groups, yielding N/A sets, where A is the size of the groups. These N/A sets are trained again in parallel to obtain N/A new sets of support vectors. This merging process is repeated, forming a reduction tree, until a single set of support vectors remains. This finishes one iteration of the algorithm. The final set of support vectors is then combined with the initial N partitions to start a

```
@task(returns=dict)
def cluster_points_sum(fragment_points, centers, ind):
    center2points = {c: [] for c in range(0, len(centers))}
    for x in enumerate(fragment_points):
        closest_center = ... # get the nearest center
        center2points[closest_center].append(x[0] + ind)
    # add up and count vectors of each center
    return partial_sum(fragment_points, center2points, ind)
```

Fig. 3. Code of `cluster_points_sum` task.

```
@task(returns=dict)
def merge_reduce_task(*data):
    reduce_value = data[0]
    for i in range(1, len(data)):
        # merge data
        reduce_value = reduce_centers(reduce_value, data[i])
    return reduce_value
```

Fig. 4. Code of the `merge_reduce_task` task.

```
def kmeans_frag(X, num_points, num_centers, dimensions, epsilon,
     max_iterations, num_fragments, seed):
    size = int(num_points / num_fragments)
    mu = init_centers_random(dimensions, num_centers, seed)
    oldmu = []
    it = 0

    while not has_converged(mu, oldmu, epsilon, it, max_iterations):
        oldmu = mu
        partialResult = []

        for f in range(num_fragments):
            partialResult.append(cluster_points_sum(X[f], mu, f * size))

        mu = merge_reduce(partialResult, chunk=50)
        mu = compss_wait_on(mu)
        mu = [mu[c][1] / mu[c][0] for c in mu]
        it += 1

    return (it, mu)
```

Fig. 5. K-means iteration code in PyCOMPSs.

new iteration. The algorithm stops when a fixed number of iterations is reached, or until convergence (i.e., when the final set of support vectors does not change significantly in two consecutive iterations). Fig. 6 shows a diagram of the training process when $A = 2$, that is, sets of support vectors are merged two by two in an inverted binary tree.

The training process of C-SVM can be easily expressed as a workflow of tasks, where each task performs the training of a subset of vectors and passes the resulting support vectors to the next task. This makes C-SVM a good candidate for being parallelized with PyCOMPSs. Our C-SVM implementation in PyCOMPSs consists of two tasks: `read_partition` and `train`. The `read_partition` task reads one partition of the input dataset from disk, and the `train` task merges together a variable number of sets of vectors and carries out the training process that yields a new set of support vectors. `train` tasks can receive a variable number of input arguments thanks to the `*args` feature. For building a decision function, `train` tasks use scikit-learn, a widely used machine learning library for Python.

Fig. 7 shows the iteration code of the C-SVM implementation in PyCOMPSs. The `_cascade_iteration` function (line 1) performs one iteration of the algorithm. This function is called iteratively until a fixed number of iterations or until convergence. The argument `partitions` is a list of future objects resulting from the `read_partition` tasks. We assume that the input dataset is divided in a collection of files, and run one `read_partition` task per file.

The first section of the iteration (lines 5–7) spawns a `train` task for each partition, and stores the results in the list q (line 7). These `train` tasks correspond to the first layer of the iteration, and train each partition merged with the support vectors of the previous iteration (line 6). In the case of the first iteration, `feedback` is empty. This means that `data` will be a list of support vectors in the first iteration and two lists of support vectors in the remaining iterations. The fact that `data` is a list of objects of

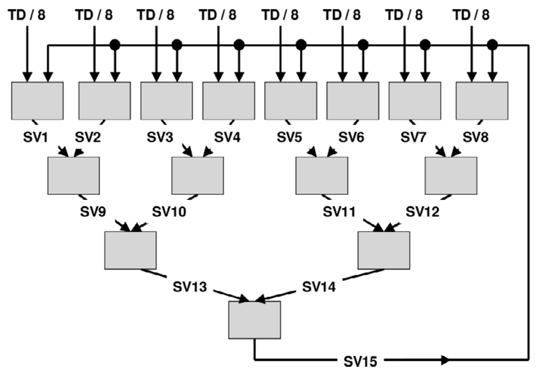


Fig. 6. Diagram of the C-SVM training process for $A = 2$ extracted from Graf et al. [41]. The support vectors obtained in one iteration (SV15) are combined with the input data for the next iteration.

unknown size is expressed with the ∗ symbol. After this section, q contains a list of future objects that represent sets of support vectors.

In the second section of the iteration (lines 10–14), the algorithm performs the reduction process. For this, we remove self._cascade_arity number of elements from q (lines 11–12), and spawn a train task that merges and trains these elements (line 14). The support vectors returned by this task are inserted in q again, and the process is repeated until q contains self._cascade_arity elements or less. Thus, self._cascade_arity controls the arity of the reduction process.

The final section of the iteration (line 17) consists of spawning the last train task, and synchronizing using the compss_wait_on method. This synchronization is required to get the actual result of the last train task and check the convergence. Note that the code is completely equivalent to a sequential implementation of C-SVM except for the compss_wait_on call and the @task annotation.

Fig. 8 shows the definition of the train task. This task simply merges together the variable number of input lists received, and carries out the training process using scikit-learn's SVC class.

5. Productivity evaluation

In this section, we estimate the productivity of PyCOMPSs by evaluating how complex are the implementations described in Section 4. Towards this, we compare the PyCOMPSs implementations to equivalent codes in MPI written with mpi4py [43], a Python wrapper for different back-end MPI versions. The MPI codes are available online, together with an script to compute the metrics used in our productivity evaluation. We compare PyCOMPSs to MPI because MPI is the most prominent general-purpose distributed programming model that can be used effectively to run data analytics algorithms in HPC clusters.

5.1. MPI implementations

We have implemented the MPI version of K-means as similar to the PyCOMPSs implementation as possible to minimize their accidental complexity [44]. However, the MPI version uses a single function instead of two to compute the distances and partial means of each cluster. The MPI version does not have a dedicated merge_reduce function because MPI provides the native functions reduce and allreduce. Using allreduce, we can add the partial results and send them to each processes, where we divide them by the total number of processes to compute the final mean (or center).

Both K-means versions generate the input dataset randomly at run time, and both versions support generating the data in two ways: centralized generation, where a single process (i.e., the master) generates all the data and sends a partition to each worker; and distributed generation, where each worker generates a partition of the input data. In the case of MPI, centralized generation is much more complex because it requires more communications, and cannot be used with large datasets because the indices of the partitions become larger than the maximum number that can be transferred (a 32-bit integer in C). We have

```
1  def _cascade_iteration(self, partitions, feedback):
2      q = []
3
4      # first layer
5      for partition in partitions:
6          data = filter(None, [partition, feedback])
7          q.append(train(False, *data, **self._clf_params))
8
9      # reduction
10     while len(q) > self._cascade_arity:
11         data = q[:self._cascade_arity]
12         del q[:self._cascade_arity]
13
14         q.append(train(False, *data, **self._clf_params))
15
16     # last layer
17     final = compss_wait_on(train(True, *q, **self._clf_params))
18
19     # check convergence
20     ...
21
22     return final[:-1]
```

Fig. 7. C-SVM iteration code in PyCOMPSs.

not implemented workarounds to this issue, as this would further increase the complexity of the MPI implementation.

The MPI version of C-SVM [25] differs significantly from the PyCOMPSs version due to MPI programming style and limitations. Nevertheless, both versions employ the same scikit-learn class to train the sets of support vectors in the reduction process. The main differences between the two versions are that the MPI implementation can only run on a power of two number of processes, that the number of partitions must be equal to the number of processes, and that the reduction is always performed in a binary tree (i.e., *arity* of two). Moreover, due to the difficulty in handling different processes, the layers of the reduction in the MPI version are synchronous. This means that all tasks in a layer need to finish before starting the next layer. Conversely, the simplicity of PyCOMPSs' programming model allows our implementation to use an arbitrary number of processes, partitions, and *arity* in the reduction process. In addition to this, since PyCOMPSs handles load balancing in a transparent manner, tasks can start executing as soon as their dependencies are available, and synchronization only happens at the end of each iteration. Implementing C-SVM in MPI with the same characteristics as in PyCOMPSs would require a significant development effort, and would produce a much more complex code.

5.2. Evaluation metrics

There are many different proposals to estimate the complexity, development and maintenance effort of a program, such as the *Constructive Cost Model* (COCOMO) [45]. However, most of these models are designed to guide the development effort in large software projects, and might give misleading results with small applications of less than 1,000 lines of code. For this reason, we have employed three simple software metrics in our evaluation: source lines of code (SLOC) [46], Cyclomatic complexity [47], and NPath [48] complexity. Each of these metrics provides different insight about the complexity of the codes.

Source lines of code is an estimation of the complexity of a given code by counting its number of lines of code. We use the most common definition of physical SLOC, which is the number of text lines of the program excluding comments.

Cyclomatic complexity is the number of linearly independent paths in a given code snippet. This represents the number of fundamental circuits in the code's flow-graph representation. Cyclomatic complexity is computed as the total number of logical conditions, such as *if* and *while* statements, plus one.

NPath complexity is the number of acyclic execution paths through a code snippet, and addresses some of the issues and limitations of Cyclomatic complexity [49]. NPath complexity is computed using the code control flow graph, where nodes are basic blocks of code or branching points, and edges represent possible execution flows. NPath complexity can be thought of as the number of possible execution combinations of a code snippet. Thus, from a code testing point of view, NPath defines the number of tests required to cover all possible outcomes.

SLOC gives a general idea of the complexity of an application as long codes are usually more complex than short codes. However, SLOC is highly affected by formatting and code style. In order to minimize this effect, we have computed SLOC using *cloc*[2] tool after formatting the algorithms with pycodestyle.[3] Nevertheless, small size differences between programs are typically not relevant.

[2] https://github.com/AlDanial/cloc.

[3] https://pypi.org/project/pycodestyle/.

```
@task(returns=tuple)
def train(return_classifier, *args, **kwargs):
    # merge args
    ...

    clf = SVC(random_state=1, **kwargs)
    clf.fit(X, y)

    # return results
    ...
```

Fig. 8. Definition of `train` task.

Cyclomatic complexity has the advantage that it is not affected by code formatting, and that it is less sensitive to code style. However, Cyclomatic complexity has also received significant criticism [49]. The main limitation of Cyclomatic complexity is that n nested *if* statements have the same complexity as n independent *if* statements. Even though the nested statements produce an exponential number of paths (2^n), while the independent statements produce a linear number of paths ($2n$). Nevertheless, we decided to include Cyclomatic complexity in our evaluation because it is still used by many tools (e.g., SonarQube[4]), and because the assumption that more control flow statements imply more complex programs is true in many cases.

NPath complexity was proposed to overcome the limitations of Cyclomatic complexity. NPath takes into account the nesting level of the code and provides, among other things, a bound for the minimum number of tests required for having a 100% code coverage. Usually, NPath complexity is considered low between 1 and 4, moderate between 5 and 7, high between 8 and 10, and extreme when higher than 10. NPath complexity is a critical metric in software development as testing can be as important as the development process itself. This is especially true in the HPC field, where programs run on large clusters for long periods of time, and buggy or untested code may result in the waste of computational resources. To compute Cyclomatic and NPath complexities we have employed sourced Babelfish tools.[5]

Finally, it is worth noting that all these metrics measure *sequential* complexity. They do not take into account parallel complexity issues. In PyCOMPSs, the user only needs to deal with sequential complexity because parallelism is handled by the runtime. Conversely, in MPI applications, users need to deal with both types of complexity.

5.3. Results

Table 1 shows the complexities of each implementation of the algorithms. We see that PyCOMPSs implementations report consistently better complexities than the MPI versions. All metrics have been computed leaving out the *main* method because we consider that the initialization and general orchestration are not part of the application itself. We compute SLOC and Cyclomatic complexity on the whole source code of the algorithms. However, NPath complexity is computed for each function, and thus we report maximum and mean value. To better understand this mean value, we also report the total number of functions and the sum of the NPath complexity of all functions.

Both K-means implementations have similar SLOC value as the application is short and has little communication. While the MPI

[4] https://www.sonarqube.org/.

[5] https://github.com/bblfsh/tools.

Table 1
Complexity metrics for the implementations of C-SVM and K-means using MPI and PyCOMPSs.

	K-means		C-SVM	
	MPI	PyCOMPSs	MPI	PyCOMPSs
SLOC	138	**118**	625	**192**
Cyclomatic	**20**	**20**	108	**23**
NPath: max	10	**6**	34	**4**
NPath: average	3.5	**2.9**	4.9	**2.5**
- NPath: sum	39	**35**	270	**35**
- Number of methods	**11**	12	55	**14**

version requires more lines for handling communications, the PyCOMPSs version requires more lines to handle the reduction. In contrast, the MPI version of C-SVM is more than three times longer than the PyCOMPSs version (625 vs. 192). This is because the C-SVM algorithm has a lot of non-symmetric communications during the reduction, and handling these communications with MPI is complex.

Both K-means implementations also have similar Cyclomatic complexity. Again, this is because there are not many communications in K-means, and the logic of the PyCOMPSs reduction compensates for the additional MPI communication management. In the case of C-SVM, however, the MPI version has a much higher Cyclomatic complexity than the PyCOMPSs implementation (108 vs. 23). This is because MPI needs to control which processes are used in every layer of the reduction. This requires a lot of conditional statements, and results in over four times more logical statements than in the PyCOMPSs version.

The method with higher NPath complexity in the MPI version of K-means (10) is the main loop that updates the new centers and sends them for the next iteration. This method orchestrates the initialization of the centers, the broadcasting of the partial sums, and the processes that compute the centers in each iteration. This results in an extremely complex method that requires at least ten test cases. Conversely, the equivalent method in PyCOMPSs has an NPath complexity of 5 because it does not need to handle communications. The method with the higher NPath complexity in the PyCOMPSs version of K-means is `cluster_points_sum` (6). On average, the MPI functions have a slightly higher NPath complexity (3.5) than the PyCOMPSs functions (2.9).

In the C-SVM case, the method with higher NPath complexity of the MPI version (34) is the function that orchestrates the cascade. The equivalent method in PyCOMPSs has a much lower NPath complexity (4). This method contains little communication in both versions, but it is 8.5 times less complex in PyCOMPSs because creating the reduction requires a single loop with no conditional statements (see Fig. 7). In contrast, the MPI implementation needs to handle the information of which is the current layer, and which subset of processes is going to process it. On average, the PyCOMPSs version of C-SVM has half the NPath complexity of the MPI version (2.5 vs. 4.9). However, the MPI implementation contains several low complexity auxiliary methods that help reduce its mean NPath complexity, and compensate for other methods that are more complex. In total, the MPI version has 8 methods with an NPath complexity of more than 10, and two methods with a complexity of more than 30. PyCOMPSs has around four times less functions, and an extremely low mean NPath complexity (2.5), with all methods in the 1 to 4 interval.

We see that MPI implementations are significantly more complex than PyCOMPSs implementations. This difference in complexity grows with the application size because MPI needs to handle each process independently. This requires additional conditional statements, and leads to an exponential increase on the number of possible execution paths. In contrast, the complexity of PyCOMPSs applications remains low and stable regardless of the application size, as PyCOMPSs handles all communication automatically. Overall, MPI codes are much more difficult to test, debug, and understand than PyCOMPSs codes.

6. Performance evaluation

In this section, we evaluate the execution performance of the PyCOMPSs implementation of K-means and C-SVM, and we compare this performance with the performance of the MPI version of the codes. We use K-means and C-SVM because these are, respectively, two of the most popular unsupervised and supervised learning algorithms [50,51].

6.1. Testbed

We run our experiments in the MareNostrum 4 supercomputer.[6] MareNostrum 4 consists of 3,456 general purpose nodes, where each node has two Intel Xeon Platinum 8160 24C at 2.1 GHz chips with 24 processors each. The nodes that we use in our experiments have 96GB of memory, run Linux operating system, and are interconnected through an Intel Omni-Path architecture.

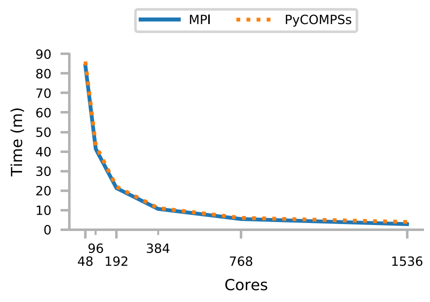
6.2. K-means

We evaluate the performance of the K-means application by analyzing execution times, strong scaling and weak scaling of both implementations (MPI and PyCOMPSs). We employ the distributed generation mechanism explained in Section 4, and always use a number of partitions equal to the number of available cores. To evaluate execution time and strong scalability, we run both versions of the algorithm using 1 up to 32 MareNostrum 4 nodes (48 to 1,536 cores), a dataset of 100 million vectors with 50 dimensions, and 50 centers. To evaluate the weak scalability of the implementations, we use a fixed problem size of 10 million vectors with 50 dimensions per node. This means that in the experiments with 32 nodes we use 320 million vectors. In all cases, we run 6 iterations of the algorithm. Fig. 9 shows the results obtained. For each experimental setting, we present the average time of five executions.

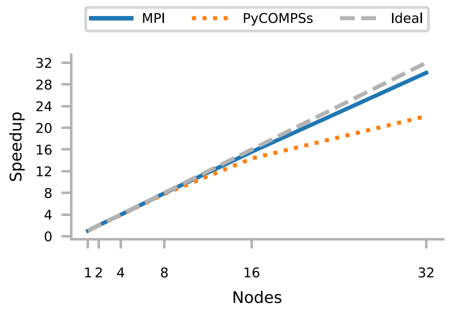
As it can be seen in Figs. 9(a) and 9(b), PyCOMPSs achieves similar execution time to MPI, and similar strong scalability with up to 16 nodes (768 cores). However, PyCOMPSs suffers a small drop in scalability caused by task scheduling overhead when using 32 nodes. Fig. 9(c) shows the scalability when increasing the input dataset proportionally with the available resources. In the ideal case, execution time should remain constant. Again, we see that PyCOMPSs achieves similar performance to MPI except for a small decrease in scalability when using 32 nodes.

PyCOMPSs overhead when using 32 nodes is caused by the scheduling of the `cluster_points_sum` tasks. K-means spawns one of these tasks per partition at the beginning of every iteration. Since we define one partition per core, this means that in the experiments with 32 nodes, PyCOMPSs needs to schedule 1,536 `cluster_points_sum` tasks in a short period of time. The delay between the scheduling of the first and the last of these tasks is $N_t \cdot T_o$, where N_t is the number of tasks and T_o is the scheduling overhead per task. This delay is of around 12 s in the experiment with 32 nodes and 10 million vectors, and of 17 s in the experiment with 32 nodes and 320 million vectors. This means that PyCOMPSs introduces an overhead of 7.79 and 11.26 ms per task respectively. This overhead could be reduced with a more efficient scheduler or by running the

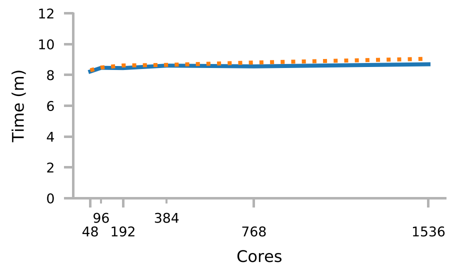
6 https://www.bsc.es/marenostrum



(a) Strong scaling execution time.



(b) Strong scaling speedup.



(c) Weak scaling execution time.

Fig. 9. Performance results for strong and weak scaling of K-means in PyCOMPSs and MPI.

Table 2
Summary of the datasets employed for the C-SVM performance evaluation.

Dataset	Vectors	Features	Description
kdd99	4,898,431	121	Intrusion detection
mnist	60,000	780	Digit recognition
ijcnn	49,990	22	Text decoding

master process in a separate node. Nevertheless, the overhead becomes less significant the longer `cluster_points_sum` tasks are. Thus, PyCOMPSs scalability improves with the granularity of the problem, which is defined by partition size, the number of dimensions, and the number of centers. The drop in scalability in Fig. 9(b) is caused by the low granularity of the tasks when using more than 32 nodes.

6.3. Cascade-SVM

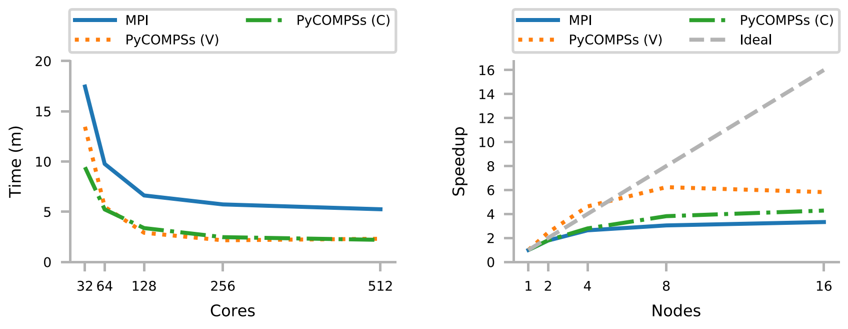
We evaluate the performance of the PyCOMPSs C-SVM implementation using three publicly available datasets[7] that are summarized in Table 2. Before running the experiments, we process the *kdd99* dataset to convert categorical features to a one-hot encoding, and we convert the *mnist* dataset to a binary problem of round digits versus non-round digits [52].

We run the PyCOMPSs and MPI versions of C-SVM using a varying number of nodes in MareNostrum. In this case, we use up to 16 nodes because the scalability of C-SVM is limited by the algorithm's design. The PyCOMPSs version can set an arbitrary number of partitions, and use an arbitrary number of cores. However, the MPI version requires the number of partitions to be equal to the number of processors, and this number to be a power of two. For this reason, we run the experiments using only 32 of the 48 cores per node. In the case of the PyCOMPSs version, we run two sets of experiments. In the first set, we use a number of partitions equal to the number of cores to better compare PyCOMPSs and MPI. In the second set of experiments, we use 512 partitions regardless of the number of cores to better understand the strong scalability of the PyCOMPSs implementation. Fig. 10 shows the results obtained. We do not present weak scaling results because we use publicly available datasets with fixed size. "PyCOMPSs (C)" and "PyCOMPSs (V)" refer to the set of experiments with constant and variable number of partitions respectively. Execution time corresponds to five iterations of the algorithm, speedup is computed using the time in one node as baseline, and results are averaged over five executions. In the training process we use an RBF kernel with $C = 10{,}000$ and $\gamma = 0.01$.

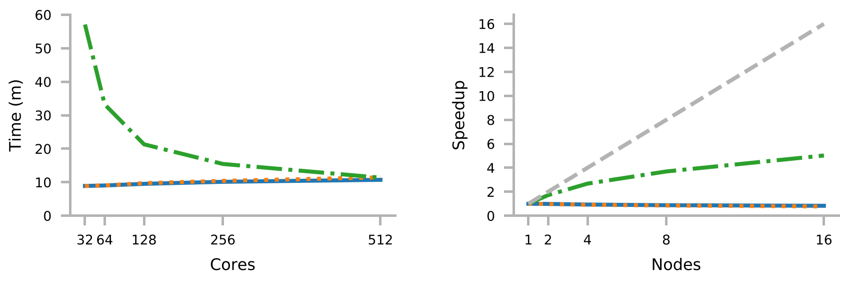
As we can see, when using a variable number of partitions, PyCOMPSs achieves similar performance to MPI, with small variations in the *kdd99* and *ijcnn* datasets. In the particular case of the *kdd99* dataset, PyCOMPSs outperforms MPI both in execution times and scalability. This means that PyCOMPSs introduces similar or less overhead than MPI in terms of communication and data transfers, as computation time in both versions of the algorithm is mainly driven by calls to the scikit-learn library.

The strong scaling speedup when using a constant number of partitions in PyCOMPSs is similar for all datasets. Results are slightly worse in the *ijcnn* case because this dataset has lower granularity. Conversely, using a variable number of partitions results in much lower scalability in all cases, both in PyCOMPSs and MPI. This is caused by the relationship between partition size and scikit-learn's training time. Increasing the number of partitions also increases the complexity of the C-SVM algorithm, and generates overhead as more tasks and data transfers need to be processed. At the same time, smaller partitions can be trained faster. This creates a trade-off between management overhead and training time that depends on the characteristics of the dataset. In the case of the *kdd99* dataset, partition size has a strong impact on training time, that is, small partitions are processed much faster than large partitions. This results in better scalability because execution time decreases as the number of

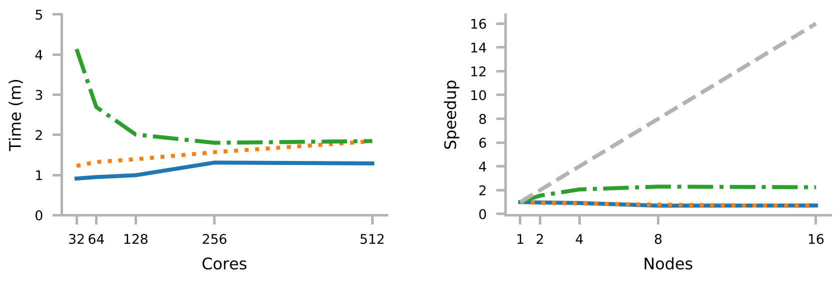
[7] Available at https://www.csie.ntu.edu.tw/~cjlin/libsvmtools/datasets/ and http://www.kdd.org/kdd-cup/view/kdd-cup-1999/Data.



(a) kdd99



(b) mnist



(c) ijcnn

Fig. 10. Execution time and speedup of C-SVM in PyCOMPSs with variable and constant number of partitions (denoted with V and C), and MPI (RBF kernel with $C = 10,000$ and $\gamma = 0.01$).

partitions increases. Conversely, in the case of the *mnist* and *ijcnn* datasets, partition size does not have a strong impact on training time. This results in poor scalability as increasing the number of partitions also increases execution time due to task and data management overhead. This trade-off affects both MPI and PyCOMPSs because both versions employ scikit-learn.

Apart from the trade-off between partition size and number of partitions, the C-SVM algorithm has limited scalability by design. The reduction process of each iteration accumulates support vectors in the lower layers (see Fig. 6). This means that as the reduction progresses, the parallelism decreases and the execution time of layers increases. This results in load imbalance as the lower layers are more computationally intensive, and in low efficiency as the maximum parallelism is only exploited in the first layer of every iteration.

6.4. Discussion

Our experiments show that PyCOMPSs achieves similar performance and scalability to MPI in most cases, and that PyCOMPSs outperforms MPI in certain situations. The experiments with K-means show that the main limitation of PyCOMPSs is the task scheduling overhead when there is a large number of ready tasks and a large number of resources. In these cases, PyCOMPSs can introduce an overhead of around 9 ms per task, which can result in delays of 12 to 17 s when scheduling 1,536 tasks. However, this overhead is negligible when processing large datasets with high granularity.

In the experiments with C-SVM, PyCOMPSs introduces similar communication overhead to MPI. However, PyCOMPSs achieves slightly higher execution times than MPI with the `ijcnn` dataset due to its low granularity. This is consistent with the behavior observed with K-means, and suggests that PyCOMPSs typically performs better with large applications and long tasks. Nevertheless, BDA applications that process large amounts of data typically run for more than 4 min, and have medium to high granularity tasks.

7. Conclusions

In this paper, we demonstrate how BDA algorithms can be developed with PyCOMPSs, and how these algorithms can be executed in an HPC environment. Moreover, we present a comprehensive evaluation of the PyCOMPSs programming model and runtime. As part of this evaluation, we compare PyCOMPSs to MPI from the point of view of their productivity and performance HPC clusters.

Our analysis shows that PyCOMPSs provides a much more productive programming model than MPI. PyCOMPSs allows developers to write BDA algorithms with more functionality and less complexity than MPI. More precisely, the analysis of the NPath complexity shows that PyCOMPSs routines remain in the 1 to 4 complexity interval, while MPI codes can reach complexities of more than 30. Moreover, MPI complexity grows extremely fast with the size of the application. This makes PyCOMPSs an **excellent** choice for developing large and complex BDA applications. In addition to this, our experimental evaluation shows that PyCOMPSs achieves similar performance to MPI in HPC environments. These results are crucial, since MPI is one of the most prominent HPC programming models, and is highly oriented to performance. MPI outperforms PyCOMPSs in certain low granularity scenarios, but PyCOMPSs can achieve similar execution times and scalability to MPI in most cases.

The work presented in this paper complements previous work that compared the performance of COMPSs and Spark in HPC environments [28]. We have shown that PyCOMPSs provides a general-purpose programming model with the best trade-off in terms of productivity, flexibility, and performance. In the future, we plan to improve PyCOMPSs' task scheduler and to further experiment with data analytics algorithms in HPC environments.

Declaration of competing interest

The authors declare that they have no known competing financial interests or personal relationships that could have appeared to influence the work reported in this paper.

Acknowledgments

This work has received funding from the European Union's Horizon 2020 research and innovation programme under the Marie Skłodowska-Curie grant agreement H2020-MSCA-COFUND-2016-754433. This work has been supported by the Spanish Government (SEV2015-0493), by the Spanish Ministry of Science and Innovation (contract TIN2015-65316-P), by Generalitat de Catalunya, Spain (contract 2014-SGR-1051). The research leading to these results has also received funding from the collaboration between Fujitsu and BSC (Script Language Platform).

References

[1] V. Turner, The digital universe of opportunities: rich data and the increasing value of the internet of things, Tech. rep., International Data Corporation, 2014.
[2] V. Raghupathi, W. Raghupathi, Big data analytics in healthcare: promise and potential, Health Inf. Sci. Syst. 2 (1) (2014).
[3] R. Dubey, A. Gunasekaran, S.J. Childe, S.F. Wamba, T. Papadopoulos, The impact of big data on world-class sustainable manufacturing, Int. J. Adv. Manuf. Technol. 84 (1) (2016) 631–645.
[4] R. Kitchin, The real-time city? big data and smart urbanism, GeoJournal 79 (1) (2014) 1–14.
[5] V. Marx, The big challenges of big data, Nature 498 (2013) 255.
[6] A. Szalay, Extreme data-intensive scientific computing, Comput. Sci. Eng. 13 (6) (2011) 34–41.
[7] G. Brumfiel, Down the petabyte highway, Nature 469 (2011) 282–283.
[8] J. Ahrens, B. Hendrickson, G. Long, S. Miller, R. Ross, D. Williams, Data-intensive science in the US DOE: case studies and future challenges, Comput. Sci. Eng. 13 (6) (2011) 14–24.
[9] C.L.P. Chen, C.-Y. Zhang, Data-intensive applications, challenges, techniques and technologies: a survey on big data, Inform. Sci. 275 (2014) 314–347.
[10] D.A. Reed, J. Dongarra, Exascale computing and big data, Commun. ACM 58 (7) (2015) 56–68.
[11] M. Zaharia, M. Chowdhury, T. Das, A. Dave, J. Ma, M. McCauley, M.J. Franklin, S. Shenker, I. Stoica, Resilient distributed datasets: a fault-tolerant abstraction for in-memory cluster computing, in: Proceedings of the 9th USENIX Conference on Networked Systems Design and Implementation, 2012, p. 2.
[12] J.L. Reyes-Ortiz, L. Oneto, D. Anguita, Big data analytics in the cloud: spark on hadoop vs MPI/OpenMP on beowulf, Procedia Comput. Sci. 53 (2015) 121–130.
[13] I.H. Witten, E. Frank, M.A. Hall, C.J. Pal, Data Mining: Practical Machine Learning Tools and Techniques, fourth ed., Morgan Kaufmann, 2016.
[14] M. Chen, S. Mao, Y. Liu, Big data: a survey, Mob. Netw. Appl. 19 (2) (2014) 171–209.
[15] W. Inoubli, S. Aridhi, H. Mezni, M. Maddouri, E.M. Nguifo, An experimental survey on big data frameworks, Future Gener. Comput. Syst. 86 (2018) 546–564.
[16] J. Dean, S. Ghemawat, Mapreduce: simplified data processing on large clusters, Commun. ACM 51 (1) (2008) 107–113.
[17] Apache Cassandra, https://cassandra.apache.org/.
[18] K. Shvachko, H. Kuang, S. Radia, R. Chansler, The Hadoop distributed file system, in: Proceedings of the 26th Symposium on Mass Storage Systems and Technologies, 2010, pp. 1–10.
[19] L. Dagum, R. Menon, OpenMP: an industry standard API for shared-memory programming, IEEE Comput. Sci. Eng. 5 (1) (1998) 46–55.
[20] W. Gropp, E. Lusk, N. Doss, A. Skjellum, A high-performance, portable implementation of the MPI message passing interface standard, Parallel Comput. 22 (6) (1996) 789–828.
[21] R. Tous, A. Gounaris, C. Tripiana, J. Torres, S. Girona, E. Ayguadé, J. Labarta, Y. Becerra, D. Carrera, M. Valero, Spark deployment and performance evaluation on the MareNostrum supercomputer, in: Proceedings of the International Conference on Big Data, 2015, pp. 299–306.
[22] E. Tejedor, Y. Becerra, G. Alomar, A. Queralt, R.M. Badia, J. Torres, T. Cortes, J. Labarta, PyCOMPSs: parallel computational workflows in python, Int. J. High Perform. Comput. Appl. 31 (1) (2017) 66–82.
[23] K.J. Millman, M. Aivazis, Python for scientists and engineers, Comput. Sci. Eng. 13 (2) (2011) 9–12.
[24] M. Götz, C. Bodenstein, M. Riedel, HPDBSCAN: highly parallel DBSCAN, in: Proceedings of the Workshop on Machine Learning in High-Performance Computing Environments, 2015, pp. 2:1–2:10.
[25] P. Glock, Design and evaluation of an SVM framework for scientific data applications (Master's thesis), Maastricht University, 2015.
[26] C.-J. Hsieh, S. Si, I.S. Dhillon, Communication-efficient distributed block minimization for nonlinear kernel machines, in: Proceedings of the 23rd ACM SIGKDD International Conference on Knowledge Discovery and Data Mining, 2017, pp. 245–254.
[27] E. Totoni, T.A. Anderson, T. Shpeisman, HPAT: high performance analytics with scripting ease-of-use, in: Proceedings of the International Conference on Supercomputing, 2017, pp. 9:1–9:10.
[28] J. Conejero, S. Corella, R.M. Badia, J. Labarta, Task-based programming in COMPSs to converge from HPC to big data, Int. J. High Perform. Comput. Appl. 32 (1) (2018) 45–60.
[29] C. Misale, M. Drocco, G. Tremblay, A.R. Martinelli, M. Aldinucci, PiCo: high-performance data analytics pipelines in modern C++, Future Gener. Comput. Syst. 87 (2018) 392–403.
[30] J.M. Wozniak, T.G. Armstrong, M. Wilde, D.S. Katz, E. Lusk, I.T. Foster, Swift/T: large-scale application composition via distributed-memory dataflow processing, in: Proceedings of the 13th IEEE/ACM International Symposium on Cluster, Cloud, and Grid Computing, 2013, pp. 95–102.
[31] F. Marozzo, D. Talia, P. Trunfio, JS4Cloud: script-based workflow programming for scalable data analysis on cloud platforms, Concurr. Comput.: Pract. Exper. 27 (17) (2015) 5214–5237.
[32] F. Pedregosa, G. Varoquaux, A. Gramfort, V. Michel, B. Thirion, O. Grisel, M. Blondel, P. Prettenhofer, R. Weiss, V. Dubourg, J. Vanderplas, A. Passos, D. Cournapeau, M. Brucher, M. Perrot, Édouard. Duchesnay, Scikit-learn: machine learning in python, J. Mach. Learn. Res. 12 (Oct) (2011) 2825–2830.
[33] W. McKinney, Python for Data Analysis: Data Wrangling with Pandas, NumPy, and IPython, first ed., O'Reilly, 2012.
[34] Redis, https://redis.io.
[35] J. Martí, D. Gasull, A. Queralt, T. Cortes, Towards DaaS 2.0: enriching data models, in: Proceedings of the 9th World Congress on Services, 2013, pp. 349–355.
[36] V. Pillet, J. Labarta, T. Cortés, S. Girona, PARAVER: a tool to visualize and analyze parallel code, in: Proceedings of the 18th World Occam and Transputer User Group Technical Meeting, 1995, pp. 17–32.
[37] Slurm Workload Manager, https://slurm.schedmd.com/.
[38] Apache Mesos, https://mesos.apache.org/.
[39] R. Amela, C. Ramon-Cortes, J. Ejarque, J. Conejero, R.M. Badia, Enabling python to execute efficiently in heterogeneous distributed infrastructures with PyCOMPSs, in: Proceedings of the 7th Workshop on Python for High-Performance and Scientific Computing, 2017, pp. 1:1–1:10.
[40] S. Lloyd, Least squares quantization in PCM, IEEE Trans. Inform. Theory 28 (2) (1982) 129–137.
[41] H.P. Graf, E. Cosatto, L. Bottou, I. Durdanovic, V. Vapnik, Parallel support vector machines: the cascade SVM, in: Proceedings of the 17th International Conference on Neural Information Processing Systems, 2004, pp. 521–528.

[42] C. Cortes, V. Vapnik, Support-vector networks, Mach. Learn. 20 (3) (1995) 273–297.
[43] L.D. Dalcin, R.R. Paz, P.A. Kler, A. Cosimo, Parallel distributed computing using python, Adv. Water Resour. 34 (9) (2011) 1124–1139.
[44] F.P. Brooks Jr., No silver bullet essence and accidents of software engineering, Computer 20 (4) (1987) 10–19.
[45] B.W. Boehm, C. Horowitz, Brown, Reifer, Chulani, R. Madachy, B. Steece, Software Cost Estimation with Cocomo II with Cdrom, first ed., Prentice Hall, 2000.
[46] V. Nguyen, S. Deeds-Rubin, T. Tan, B. Böhm, A SLOC counting standard, in: The 22nd Annual Forum on COCOMO and Systems/Software Cost Modeling, 2007.
[47] T.J. McCabe, A complexity measure, IEEE Trans. Softw. Eng. SE-2 (4) (1976) 308–320.
[48] B.A. Nejmeh, NPATH: a measure of execution path complexity and its applications, Commun. ACM 31 (2) (1988) 188–200.
[49] M. Shepperd, A critique of cyclomatic complexity as a software metric, Softw. Eng. J. 3 (6) (1988) 30–36.
[50] A.K. Jain, Data clustering: 50 years beyond k-means, Pattern Recognit. Lett. 31 (8) (2010) 651–666.
[51] D.R. Amancio, C.H. Comin, D. Casanova, G. Travieso, O.M. Bruno, F.A. Rodrigues, L. da Fontoura Costa, A systematic comparison of supervised classifiers, PLOS ONE 9 (4) (2014) 1–14.
[52] K. Zhang, L. Lan, Z. Wang, F. Moerchen, Scaling up Kernel SVM on limited resources: a low-rank linearization approach, in: Proceedings of the Fifteenth International Conference on Artificial Intelligence and Statistics, 2012, pp. 1425–1434.

Javier Álvarez Cid-Fuentes is a researcher at the Workflows and Distributed Computing group of the Barcelona Supercomputing Center. His research interests include anomaly detection in distributed systems, as well as parallel programming models for distributed infrastructures. Álvarez Cid-Fuentes received a Ph.D. in computer science from the University of Adelaide in 2018.

Pol Álvarez Vecino is a data science master student at the Polytechnic University of Catalonia, and currently works at the Barcelona Supercomputing Center in the Workflows and Distributed Computing group. His research interests are in the field of artificial general intelligence both in terms of novel models and the distributed systems required to power them.

Ramon Amela Milian is a Master Student in Research and Innovation in Informatics - Data Science (MIRI - UPC). His main interests are HPC applications and workflow scheduling policies. At BSC, he has worked developing linear algebra kernels, machine learning algorithms, and bioinformatics applications. Also, he has contributed to the development of the COMPSs scheduling engine.

Kuninori Ishii received his Master degree in the field of information science and technology from the University of Tokyo. Ishii has been making his research in Fujitsu Ltd. Recently, he is developing a script language platform for supercomputers in the Next-Generation Technical Computing Unit of Fujitsu.

Dr. Rafael K. Morizawa has received his Ph.D. in Computer Science from the Tokyo Institute of Technology in 2001. After working for a few years in a semiconductor technology research company, he joined Fujitsu, LTD. in 2003, first working in the Laboratory on the field of semiconductor design verification. Since 2014 Dr. Morizawa is with Fujitsu's Technical Computing Solutions Unit working on the field of High-Performance Computing.

Rosa M. Badia holds a Ph.D. on Computer Science (1994) from the Technical University of Catalonia (UPC). She 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 Cientificas (CSIC). Her current research interest are programming models for complex platforms (from multicore, GPUs to Grid/Cloud). The group lead by Dr. Badia has been developing StarSs programming model for more than 10 years, with a high success in adoption by application developers. Currently the group focuses its efforts in PyCOMPSs/COMPSs, an instance of the programming model for distributed computing including Cloud, and its application to parallelize Big Data and Analytics. Dr Badia has published near 200 papers in international conferences and journals in the topics of her research. She has been very active in projects funded by the European Commission and in contracts with industry (IBM and Intel).