The Journal of Supercomputing (2024) 80:11741–11761
https://doi.org/10.1007/s11227-023-05883-z

Boosting HPC data analysis performance with the ParSoDA-Py library

Loris Belcastro[1,3] · Salvatore Giampà[3] · Fabrizio Marozzo[1,3] · Domenico Talia[1,3] · Paolo Trunfio[1,3] · Rosa M. Badia[2] · Jorge Ejarque[2] · Nihad Mammadli[2]

Accepted: 23 December 2023 / Published online: 2 February 2024

Abstract
Developing and executing large-scale data analysis applications in parallel and distributed environments can be a complex and time-consuming task. Developers often find themselves diverted from their application logic to handle technical details about the underlying runtime and related issues. To simplify this process, ParSoDA, a Java library, has been proposed to facilitate the development of parallel data mining applications executed on HPC systems. It simplifies the process by providing built-in scalability mechanisms relying on the Hadoop and Spark frameworks. This paper presents ParSoDA-Py, the Python version of the ParSoDA library, which allows for further support of commonly used runtimes and libraries for big data analysis. After a complete library redesign, ParSoDA can be now easily integrated with other Python-based distributed runtimes for HPC systems, such as COMPSs and Apache Spark, and with the large ecosystem of Python-based data processing libraries. The paper discusses the adaptation process, which takes into consideration the new technical requirements, and evaluates both usability and scalability through some case study applications.

Keywords Big data analysis · Parallel computing · HPDA · PyCOMPSs · Spark · HPC

1 Introduction

Writing and running big data analysis applications in highly parallel and distributed environments is often a challenging job. Developers often have to distract themselves from application logic to focus on defining technical details about the underlying runtime and related issues. Many researchers are working on designing and implementing tools and algorithms to extract useful information from huge amounts of data [1]. In such cases, the use of parallel and distributed data analysis techniques, frameworks (e.g., Hadoop or Spark) is crucial to handle the size and

Extended author information available on the last page of the article complexity of the data being analyzed. However, many users find it challenging to utilize such solutions for addressing big data challenges due to the programming skills required to implement appropriate data analysis methods on complex distributed systems and runtimes. ParSoDA (parallel social data analytics) [2] is a Java-based programming library for simplifying the development of parallel data and social media mining applications executed on HPC systems. Differently from other existing systems, ParSoDA was specifically designed to implement parallel scalable data analysis applications, with a particular focus on data gathered from social media. It provides scalability mechanisms based on two of the most popular parallel processing frameworks, Hadoop and Spark, which are fundamental to provide efficient and scalable services as the amount of data to be managed grows [3]. However, to further improve the library's capabilities in terms of usability, ease of programming, and support for further runtimes and libraries, a new version of ParSoDA, based on Python, is of strategic interest. Python is known for its simple and readable syntax, making it accessible to both beginners and experienced programmers. Moreover, a rich ecosystem of data analysis libraries is available in Python, such as NumPy and Pandas, which significantly expedites writing complex operations with minimal code.

This paper presents ParSoDA-Py, the Python version of the ParSoDA library, which enables the execution of ParSoDA-based applications on various Python-based distributed runtimes designed for HPC systems, such as COMPSs and Apache Spark, via PyCOMPSs and PySpark, respectively. Making the ParSoDA library available in Python involved addressing new technical requirements and considering some use cases discussed in the previous scientific papers that describe the library [2].

The contents of this paper are organized as follows. Section 3 presents an overview of the ParSoDA library. Section 4 discusses the implementation of ParSoDA on top of PyCOMPSs. Section 5.3 presents performance results of ParSoDA on PyCOMPSs and compares them with performance of ParSoDA on Spark. Finally, Sect. 6 concludes the paper.

2 Related work

Research in the field of frameworks and libraries aimed at improving the development of data analysis applications for high-performance computing (HPC) environments is a popular and evolving topic. Most current solutions are often limited to certain application contexts and, therefore, adaptable to solving only specific types of problems. Some recent surveys have attempted to provide a broad and up-to-date overview of such frameworks and libraries, with particular focus on efficient processing of large amounts of data. [3–5].

As data to be processed grow exponentially, organizations and researchers face increasing challenges in terms of computing capabilities. This requires the utilization of high-performance computing resources, such as multi-core systems, cloud infrastructure, and multi-cluster configurations, coupled with parallel and distributed algorithms. This strategic approach ensures reasonable response times in the face of this data deluge [6].

In this context, many researchers are actively engaged in developing tools and algorithms aimed at extracting valuable insights from the large amounts of data, like those coming from social media platforms. Some research projects consider not only the data analysis task, but also procedures including data processing tasks needed for building social data applications. In particular, these projects aim at helping scientists to implement all the steps that compose social data mining applications without the need to implement common operations from scratch. One notable framework in this domain is SOCLE [7], designed to optimize data preparation for social data analysis applications. It includes a versatile three-tier architecture, an algebraic framework, and a domain-specific language tailored for defining data preparation operations in social applications. For instance, SOCLE provides users with operators for data pruning, data enrichment, and data transformation and normalization. Although SOCLE's utility has been demonstrated in social applications like recommendation and analytics, there is a lack of studies assessing its scalability, and detailed framework requirements remain undisclosed. In another vein, Cuesta et al. [8] introduced a framework aimed at simplifying the extraction and analysis of Twitter data, where developers can expand its functionality by creating additional aggregation tasks using Python's MapReduce capabilities. The framework also offers prebuilt modules for carrying out sentiment analysis and generating reports. SODATO (SOcial Data Analytics Tool) [9] represents an online tool tailored for programming data analytics tasks involving social data. It leverages APIs from social media platforms to gather data, followed by a combination of web and console applications for batch-based preprocessing, data aggregation, and data analysis. In particular, SODATO provides methods for different types of analysis, including sentiment analysis, keyword analysis, content performance analysis, and social influence analysis. Zhou et al. [10] proposed a general unsupervised framework engineered for discovering events in Twitter datasets. The framework involves a pipeline process encompassing filtering, extraction, and categorization phases. During the filtering stage, a lexicon-based approach selects tweets relevant to events. Subsequently, events are extracted from these filtered tweets and grouped into categories using an unsupervised Bayesian model named the latent event & category model (LECM). You et al. [11] introduced a cloud-based framework tailored for developing social data analysis applications, with a focus on supporting smart cities and smart mobility. This comprehensive framework consists of five key components: data collection, data preprocessing, data analysis, data presentation, and data storage. It facilitates data collection from various sources, including social media platforms (e.g., Twitter and Foursquare) through their public APIs, as well as other internet sources like websites, blogs, and files. The data preprocessing component offers functions for data cleansing, filtering, and normalization. Subsequently, the data analysis component equips users with essential analysis methods, such as K-Means, DBSCAN, and self-organizing map, to facilitate data analysis.

Many other research efforts focused on analytics frameworks for scientific data, leveraging large-scale parallel computing approaches. PyOphidia [12] is a Python library designed to offer a high-level and programmatic interface for carrying out large-scale data analytics on large multi-dimensional scientific datasets. It represents the Python binding of the Ophidia framework [13], an open-source software framework designed for the management and analysis of large-scale scientific data, particularly multidimensional datacubes. Additionally, it supports workflow execution on HPC systems, also facilitates integration with well-established modules from the Python scientific ecosystem. FastFlow [14] and DSParLib [15] both serve as high-level C++ libraries for parallel programming, emphasizing pattern-based approaches. While DSParLib introduces new implementations for the pipeline and farm patterns in stream processing, FastFlow provides a broader range of patterns suitable for different computation types. It is worth noting that FastFlow supports multi-node systems through ZeroMQ and process creation, whereas DSParLib utilizes MPI for these purposes. GrPPI [16] is another C++ library focusing on composable and generic interfaces for parallel patterns, which has been widely explored for distributed computing using MPI. Programmers need only implement parallel patterns once, choosing the desired runtime at compile time.

In contrast to the systems mentioned above, ParSoDA-Py distinguishes itself by being specifically engineered to ease the implementation of big data analysis applications for execution on both HPC systems and cloud. Its features allow the efficient and scalable provision of data processing services, especially as the volume of data to be managed expands. In particular, the execution logic of ParSoDA-Py is completely abstracted from the execution environment, allowing it to potentially be used on top of a wide range of runtimes.

3 ParSoDA

ParSoDA (parallel social data analytics) [2] is a programming library originally developed for simplifying the development environment of parallel social media mining applications executed on high-performance computing systems. However, in its current version it can be used for designing general data analysis and machine learning applications on HPC systems. To achieve this goal, ParSoDA provides a set of widely used functions for processing and analyzing data collected from social media and other sources, which can be used to extract useful knowledge and patterns (e.g., topics trends, user mobility, user opinions). ParSoDA defines a general framework for a data analysis application that includes a number of steps (i.e., data acquisition, filtering, mapping, partitioning, reduction, analysis, and visualization), and provides a predefined (but extensible) set of functions for each data processing step. Thus, an application developed with ParSoDA is expressed by a concise code that specifies the functions invoked at each step. For each of these steps, ParSoDA provides a predefined set of functions. Users are free to extend this set with their own functions. For example, for the data acquisition step, ParSoDA provides crawling functions for gathering data from different source and from some of the most popular social networks (Twitter and Flickr), while for the data filtering step, ParSoDA provides functions for filtering geotagged items based on their position, time of publication, and contained keywords.

Figure 1 illustrates the execution flow of ParSoDA on a Spark cluster, composed of one or more master nodes and multiple slave nodes. Specifically, the main steps are executed within two Spark stages that run on a set of worker nodes. A stage is a set of independent tasks executing functions that do not need to perform data shuffling (e.g., transformation and action functions). Specifically: data filtering and mapping are executed within the first stage (Stage 0), while data partitioning and reduction are executed within the second stage (Stage 1). Concerning the remaining steps (data acquisition, data analysis, and data visualization), they are not strictly dependent on Spark. These steps have the flexibility to be executed either in parallel on multiple worker nodes or locally by the master node(s). It is important to note that executing them locally does not imply sequential execution, as the master node can leverage alternative parallel runtimes such as MPI to perform parallel computations.

4 ParSoDA-Py

In previous works, we demonstrated that ParSoDA is suitable for running scalable data analysis applications on cloud and HPC systems exploiting both Apache Hadoop [17] and Spark [18]. Starting from the Java version of ParSoDA, a complete redesign of the library has been done by adopting a bridge design pattern for supporting different execution runtimes and data crawlers. The design took into account new technical requirements that further simplify the development of data analysis applications, taking into consideration the dynamically typed capabilities of Python. This is achieved by structuring the applications as sequences of computation blocks.

ParSoDA-Py has been developed according to the open-closed principle (OCP) [19], which is one of the core concepts of the software design patterns. This design approach ensures that the core functionality of the library remains unaltered, maintaining stability and reliability, while simultaneously allowing for the extension of its capabilities to accommodate different execution runtimes. In particular,

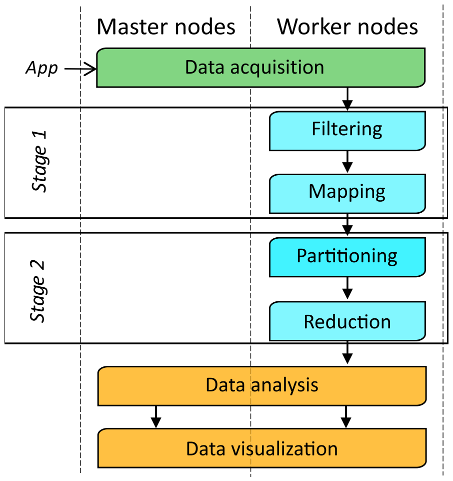


Fig. 1 Execution flow of ParSoDA on Apache Spark

ParSoDA-Py provides a trade-off between an immutable core and the flexibility needed for seamless integration with evolving technologies and runtimes. The Java version of ParSoDA has been implemented in accordance with the same principle, but with a less efficient outcome, as it did not foresee the possibility of having a unified core code shared across different runtimes. Indeed, the two existing Java versions of ParSoDA, supporting Hadoop and Spark, were developed and compiled specifically for the runtime on which they were executed. In contrast, ParSoDA-Py introduces a more flexible and efficient approach, enabling the reuse of the core components and the possibility of defining specific drivers to support new runtimes. This feature not only enhances flexibility but also streamlines the development and maintenance process. Additionally, compared to the previous version of the library, ParSoDA-Py has introduced support for distributed crawling to achieve better performance when retrieving data from remote sources.

The usability of ParSoDA-Py is significantly better than that of ParSoDA, especially since adopting the more concise syntax of the Python language makes it easier to develop an application. Differently from the Java version of ParSoDA, where workflow steps are executed directly on the Apache Hadoop/Spark environment, ParSoDA-Py delegates the execution to an abstract driver. This introduces a new level of abstraction that separates the application code from the underlying execution runtime, ensuring greater usability (e.g., the same code can be executed on a different runtime by simply changing the driver in use) and enhanced compatibility with an extensible set of runtimes.

In the remainder of this section, design and implementation details, along with innovative features, of ParSoDA-Py will be discussed. In particular, Sect. 4.1 describes the architectural design of ParSoDA-Py, Sect. 4.2 presents detailed insights into the integration process with PyCOMPSs, and Sect. 4.3 discusses the features of the data crawling component.

4.1 Design of the architecture

ParSoDA-Py retains the same main concepts of ParSoDA, but having been redesigned for Python, it needed to improve some features, such as data acquisition mechanisms. This step is the main difference between the execution flow of ParSoDA and ParSoDA-Py, which is specified further below. Another important difference is that a developer must define the initial number of partitions or partition size. This initial partitioning has the effect of parallelizing the reading of data with some specialized crawlers. As shown in Fig. 2, for each block, developers are required to specify one or more functions. Specifically, ParSoDA-Py defines a general structure for a social data analysis application that is composed by the following steps:

- *Data acquisition* it is possible to run multiple crawlers in parallel; the collected social media items are parsed and converted in an internal format suitable to be processed by the next steps. The driver of a specific runtime will take care, under the hood, of partitioning the data into chunks and distributing them to workers for parallel processing. ParSoDA-Py also introduces *Parser* classes to convert

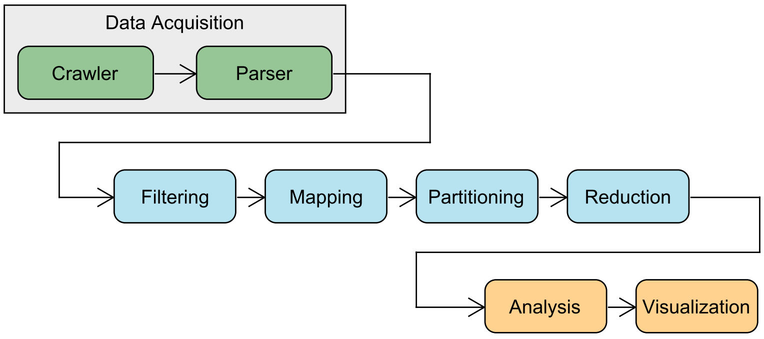


Fig. 2 Execution flow of ParSoDA-Py

social data items into a structured format. These classes can be easily created or extended by programmers, enabling them to simultaneously read data from various sources and transform it into a standardized object format.

- *Data filtering* this step filters the social media items according to a set of filtering functions.
- *Data mapping* this step transforms the information contained in each social media item by applying a set of map functions.
- *Data partitioning* during this step, data are partitioned into shards by a primary key and then sorted by a secondary key.
- *Data reduction* this step aggregates all the data contained in a shard according to the provided reduce function.
- *Data analysis* this step analyzes data using a given data analysis function to extract the knowledge of interest.
- *Data visualization* at this final step, a visualization function is applied on the data analysis results to present them in the desired format.

Figure 3 shows a UML diagram of the main components used to define an application in ParSoDA-Py. In particular, the diagram highlights *SocialDataApp*, one of the most important classes of the library, and its setter methods that must be used to build a social data analysis application. All other classes are the main components that define each step of the ParSoDA workflow, except the ParsodaDriver, which is instead used for executing the ParSoDA workflow on the underlying runtime environment.

In order to define a new application, a programmer must create a ParsodaDriver instance and a SocialDataApp object using it. The ParsodaDriver class is responsible for interfacing with the ParSoDA application workflow within the chosen underlying runtime environment. The programmer defines a ParSoDA application in a descriptive way, specifying its components (such as crawlers, filters, mapper, and reducer). Subsequently, the SocialDataApp object is employed to execute the workflow presented in Fig. 1. This execution involves translating it into a sequence of invocations of the ParsodaDriver methods, which then execute these steps within the underlying environment.

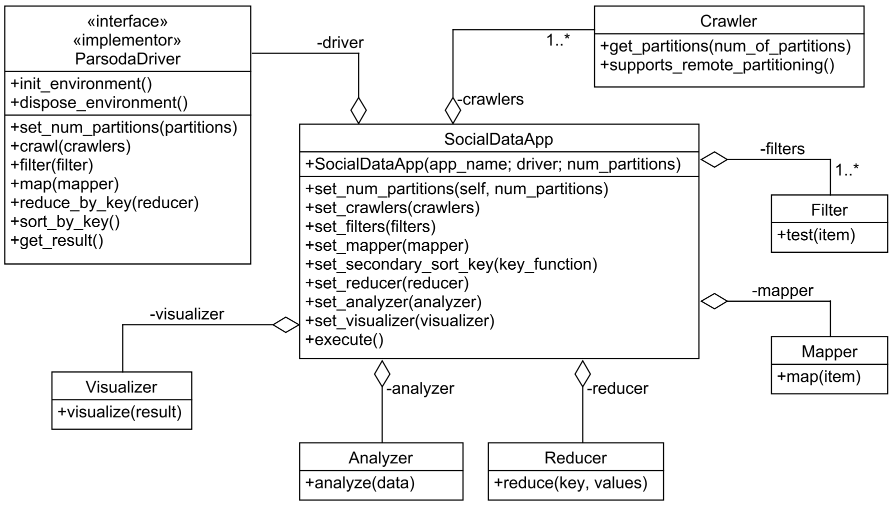


Fig. 3 Class diagram of the main ParSoDA-Py components

Figure 4 shows the bridge design pattern used for implementing multiple execution environments for ParSoDA. It defines an abstraction, the *SocialDataApp* class, which defines the high-level operations of a social data analysis application, and an implementor, the *ParsodaDriver* class, which provides the low-level operations. A valid instance of *ParsodaDriver* must provide the implementation of some methods that grant access to some parallel patterns, such as *flatMap*, *filter* and *groupByKey*, a method for getting the final result (i.e., *collect*), and some other functions for setting up and down the runtime (e.g., *init_environment*, *dispose_environment*). The

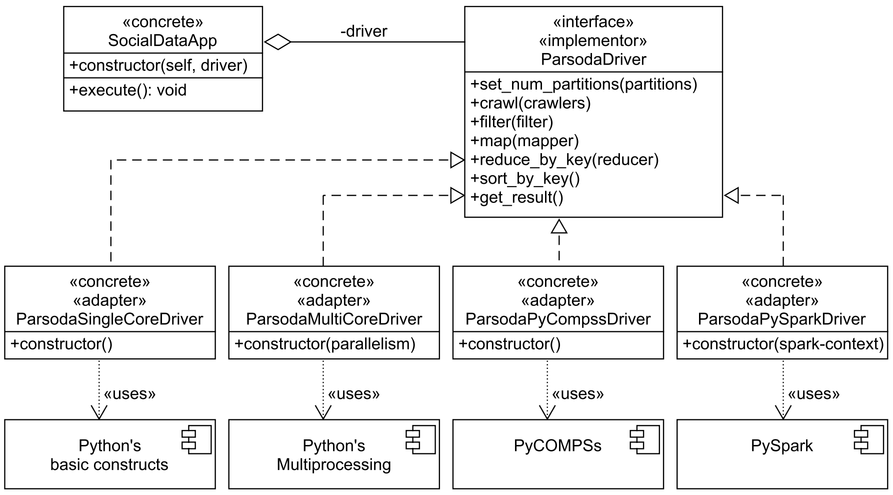


Fig. 4 Bridge pattern used to specify the execution environment for a ParSoDA-Py application

SocialDataApp class is specifically designed to effectively use such parallel patterns for executing ParSoDA applications. It is worth noting that the execution flow of a ParSoDA application is runtime-agostic, which means it remains unchanged even when changing the execution environment. Furthermore, integration with new distributed systems is facilitated. In particular, ParSoDA-Py provides four execution drivers:

- *ParsodaSingleCoreDriver* a driver that implements parallel patterns as simple sequential algorithms on a single core, on the local machine, which can be useful for testing purposes.
- *ParsodaMultiCoreDriver* it runs the application locally, based on Python's Thread Pools.
- *ParsodaPySparkDriver* it runs the application on a Spark cluster through the PySpark library.
- *ParsodaPyCompssDriver* it runs the application on a COMPSs cluster by exploiting the PyCOMPSs binding to gain access to the COMPSs runtime.

4.2 Integration with PyCOMPSs

The reason for utilizing Python in statistical analysis applications is its increasing relevance in the field. Python is supported by several robust libraries that are valuable for statistical purposes, narrowing the gap with dedicated analysis tools like Matlab and R. To ensure its usability in large projects, appropriate parallelization techniques should be incorporated.

PyCOMPSs [20] is the Python binding of COMPSs, a task-based programming environment that facilitates the development of parallel computational workflows in Python. In this approach users program their Python scripts in a sequential fashion and annotate the functions to be run as asynchronous parallel tasks. A runtime system is in charge of exploiting the inherent concurrency of the script, detecting the data dependencies between tasks and spawning them to the available resources. PyCOMPSs supports the execution of the Python applications in distributed computing platforms, including large clusters, clouds, and container-managed clusters. Applications are deployed following the master-worker paradigm, with the main script and the COMPSs runtime running in one node. The runtime starts the execution of the sequential script, and at each invocation of an annotated function, a node is added to a task-dependency graph. In this graph, nodes denote tasks and edges denote data dependencies between tasks that are identified by the runtime based on hints available in the annotations. The COMPSs runtime takes care of the orchestration of the whole application: task scheduling, resource allocation, data transfers between nodes when needed, and so on.

PyCOMPSs include the distributed dataset (DDS) [21], a lightweight library providing an interface that can be used by programmers for loading data from basic Python data structures, generators, or files, distributing the data on available nodes, and running some of the most common big data operations on it. To take advantage of DDS, the user should first load the data to a new instance of it. Once one of the *load* functions is called, the data will be partitioned and sent to the available nodes. Subsequently, the user can perform any of DDS operations to manipulate the data by invoking methods on the DDS instance. In the DDS environment, the initial data are always distributed on an arbitrary number of partitions and passed from one task to another as future objects until the programmer *synchronizes* or *collects* it. Additionally, it is possible to create a new DDS with a list of future objects from user-defined functions or send data from a DDS instance to other user-defined functions as future objects without retrieving it on the master node. This flexibility gives programmers an opportunity to use DDS methods anywhere in the code, mixing the data from those methods with their own functions without sticking to predefined data operations, as well as replacing some methods with DDS ones on an existing project. Specifically, DDS provides a range of parallel and distributed data operators, including *load*, *filter*, *map*, *group_by_key*, and *collect*.

ParSoDA-Py has been extended to support DDS, ensuring the effective integration of the ParSoDA data partitioning model with that provided by PyCOMPSs. Specifically, the integration with PyCOMPSs has been done by defining a new driver class, namely *ParsodaPyCompssDriver*, which implements different data transformer methods such as *filter()*, *flatmap()*, and *group_by_key()*, whose executions are delegated to a DDS object instantiated transparently during the initialization of the environment. Each transformer is applied to data partitions provided by crawlers, as described in Sect. 4.3.

4.3 Data crawling

The Java version of ParSoDA was initially designed to read files from HDFS, the local file-system or to collect data from specific social media (i.e., Flickr and Twitter) using predefined crawlers. However, it does not provide the possibility of exploiting other forms of parallel data reading from one or more remote sources, which can significantly limit performance in several application cases. The data crawling component of ParSoDA-Py has been completely redesigned to overcome this limitation, introducing a level of abstraction in data management that improves its flexibility. In ParSoDA-Py, to read one or more data sources, a developer must provide one or more instances of concrete subclasses derived from the crawler abstract class. Specifically, there are two different types of crawlers that can be defined:

- *Local crawler* it directly reads the data source from the node on which the application is launched, which is typically the master node. Subsequently, the read data are appropriately partitioned among the nodes of the underlying execution environment. The method of partitioning these data must be specified by the driver associated with the execution environment. Additionally, the application developer can choose the number of partitions by providing an optional parameter to the constructor of the *SocialDataApp*.
- *Distributed crawler* it performs source partitioning, enabling the provision of iterable objects. Each iterable object refers to a specific remote partition of the source upon request. For example, if the source is an HTTP endpoint, a crawler

can be defined to utilize paging and provide an iterable object for reading each page. In this scenario, the *ParsodaDriver* is capable of sending each iterable object to a specific node within the system, facilitating parallel reading of the data source.

As illustrated in Fig. 5, when developing a new crawler, developers have to implement the *get_partitions* abstract method of the *Crawler* class. This method is responsible for providing one or more source partitions, represented as iterable objects. The method also accepts a parameter to suggest the number of partitions. However, it is important to note that the crawler is not obligated to strictly adhere to this partitioning constraint. It has the flexibility to return a different number of partitions, either fewer or more, than the specified value. Each partition is an iterable object that provides instances of the *SocialDataItem* class.

To facilitate the reading of social data items from local files on the master node, ParSoDA-Py provides the *LocalFileCrawler* class. This class is an implementation of the abstract *MasterCrawler* class, which is a non-distributed crawler extending the more generic Crawler class. The *LocalFileCrawler* class requires it to be initialized by specifying an instance of the *Parser* class. A *Parser* is an invocable object designed to process a line of text and return a standard *SocialDataItem* object.

5 Study case applications

This section discusses the use of the ParSoDA-Py library for developing data analysis applications that can be executed on supported runtimes through their Python bindings. Specifically, the latest version of ParSoDA-Py offers compatibility with two popular runtimes, COMPSs and Spark, leveraging the PyCOMPSs and PySpark bindings, respectively. We carried out a large set of experiments to evaluate usability and performance of ParSoDA-Py on two data analysis applications that process data published

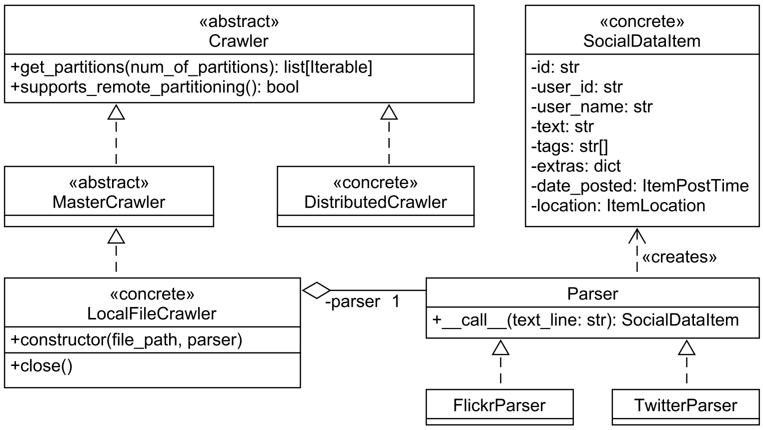


Fig. 5 APIs used for defining new crawlers and new parsers for social data items

on Twitter. The first application aims at discovering sequential patterns in user movements, so as to find the most common routes followed by users. On the other hand, the second application focuses on discovering the sentiment of social media posts by analyzing its textual content. The analysis was carried out by analyzing a large dataset of social media posts that refer to the center of Rome.

The remainder of this section is organized as follows. Section 5.1 describes the code of the trajectory mining application, built with ParSoDA-Py to run on top of both COMPSs and Spark. Section 5.2 presents the implementation of the sentiment analysis application, which analyzes a large dataset of 180 GB of social media posts from Twitter. Finally, Sect. 5.3 discusses scalability tests for evaluating turnaround time and speed-up.

5.1 Sequential pattern mining

This section discusses how to write an application that processes social media data to extract the sequential pattern in user movements, which represents the trajectories followed by users among popular places-of-interest (PoIs) in the center of Rome. Initially, we will develop an application for execution on COMPSs. Subsequently, we will demonstrate that running the same application on Spark is a straightforward process, requiring only the utilization of a different driver class. It is worth noting that ParSoDA-Py's functions are intentionally designed to be agnostic about the underlying runtime, ensuring compatibility and ease of transition between different execution environments.

Listing 1 illustrates the main function of a sequential pattern mining application, built using ParSoDA-Py. To begin, a *ParsodaPyCompssDriver* is created (line 1), which is why the application is designed for execution on COMPSs through PyCOMPSs. In particular, the PyCOMPSs driver does not require any parameters since PyCOMPSs is externally configured by invoking the *runcompss* command-line tool. Once the driver has been created, a *SocialDataApp* object is defined, passing as arguments the application's name and the driver (line 2). Optionally, the programmer can also specify the desired number of partitions to be created when loading data. During the crawling phase, a dataset containing social media posts, collected in the area around Rome, is loaded. (line 3). This dataset is stored in a network file storage accessible to all workers in the COMPSs cluster. To reduce data access latency, a *DistributedFileCrawler* is used. As described in the previous section, this specialized crawling class facilitates reading the source file, dividing it into chunks, and assigning the charge of loading the data chunks to the respective worker responsible for processing them. Consequently, the master node no longer needs to distribute the data among workers, as each worker independently retrieves the necessary data over the network. In such a way, it is possible to reduce the load of the master node and significantly improve data transfer times.

```
1  driver = ParsodaPyCompssDriver()
2  app = SocialDataApp("Sequential Pattern Mining", driver)
3  app.set_crawlers([DistributedFileCrawler('Twitter.json', TwitterParser())])
4  app.set_filters([IsGeotagged()])
5  app.set_mapper(FindPoI("RomeRoIs.kml"))
6  app.set_secondary_sort_key(lambda x: x[0])
7  app.set_reducer(ReduceByTrajectories(3))
8  app.set_analyzer(GapBIDE(0.001, 0, 10))
9  app.set_visualizer(SortGapBIDE('trajectory_mining.txt', 'support', mode='descending'
       , min_length=3))
10 app.execute()
```

Listing 1 Trajectory mining application running on COMPSs

During the filtering phase, a function, namely *isGeotagged*, is used to filter out posts having valid geotagged information (line 4). ParSoDA-Py library also provides the possibility of passing a sequence of filtering functions, which are subsequently chained to apply multiple filters to the input data.

The map function *FindPoI* (line 5) converts each geotagged post into a tuple $\langle userId, \langle datetimePost, PoI\rangle\rangle$, where *userId* is the unique user identifier, *datetime-Post* is the creation timestamp of the post, and *PoI* is the name of the Point-of-Interest (PoI) in which the post have been created. Generally, PoIs refer to tourist attractions, such as monuments, squares or bridges, or to business places, such as airports, shopping malls or train stations. A user trajectory can be represented as a sequence of PoIs visited by a user. For analyzing users' behavior, it is useful to understand whether a user visited or not a PoI. Since information on a PoI is generally limited to an address or to GPS coordinates, it is hard to match trajectories with PoIs. For this reason, it is useful to define the so-called regions-of-interest (RoIs) that represent the boundaries of the PoIs' area [22]. For these reasons, the function FindPoI receives in input a file (*RomeRoIs.kml*), containing the list of the RoIs of the most popular PoIs in Rome.

To extract the temporally ordered sequence of PoIs visited by a single user, the tuples must be aggregated by user id and sorted by timestamp. Taking as input the output of the mapping phase, the partitioning phase applies secondary sort function (line 6) to aggregate tuples by user id, i.e., the primary key, and then sort tuples in each group by timestamp, i.e., the secondary key.

During the reduction phase, a specific reduce class called *ReduceByTrajectories* (line 7) is used to transform all the social media posts of a single user into a list of daily trajectories across PoIs. This class is designed to accept a single parameter, which represents the minimum trajectory length to be taken into account. Subsequently, all the trajectories computed during the reduction phase are gathered and forwarded to the subsequent phase for analysis.

The Gap-BIDE [23] algorithm is used as a data analysis function (line 8). This algorithm is a sequential pattern mining algorithm, which takes as input a collection of sequences and mines frequent sequences. The data analysis class has three parameters, which are the minimum support, the minimum and the gap. For the data visualization phase, the *SortGapBIDE* class is specified to perform the data visualization function (line 9). The class receives three parameters: the input dataset containing user trajectories, the sort direction (descending order), and the minimum length of trajectories to be produced in output.

Finally, the application is submitted to the COMPSs cluster by invoking the *execute* method on the *SocialDataApp* object we have defined (line 10).

5.2 Sentiment analysis

Listing 2 shows the same approach proposed by Chin et al. [24], we tested the Python implementation of the ParSoDA library through a sentiment mining application, which determines the sentiment (e.g., positive or negative) of each post according to the emojis it contains. Emojis are picture characters or pictographs that originated on Japanese mobile phones in the late 1990s but gained worldwide popularity in text-based communication with the introduction of smartphones supporting input and rendering of emoji characters. By analyzing the sentiment of 1.6 million human-annotated tweets, Noval et al. [25] constructed an emoji sentiment lexicon called the Emoji Sentiment Ranking (ESR). This lexicon includes the most frequently used emojis and their corresponding attributes. The attributes encompass the emoji's pictorial representation, occurrences, Unicode code point, Unicode name, negativity, neutrality, and positivity regressed with position, as well as its sentiment score. To calculate the sentiment score of each post, we summed the ESR scores of the emojis it contains. The resulting calculation determined the post's sentiment: positive if the total score is greater than 0, negative if it is less than zero, and neutral if it is zero [26].

```
driver = ParsodaPyCompssDriver()
app = SocialDataApp("Sentiment analysis", driver)
app.set_crawlers([DistributedFileCrawler('Twitter.json', TwitterParser())])
app.set_filters([HasEmoji()])
app.set_mapper(ClassifyByEmoji("emoji.json"))
app.set_reducer(ReduceByEmojiPolarity())
app.set_analyzer(TwoFactionsPolarization())
app.set_visualizer(PrintEmojiPolarization('emoji_polarization.txt'))
app.execute()
```

Listing 2 Sentiment analysis application running on COMPSs

It is worth noting that ParSoDA-Py significantly simplifies the development process, leading to a very concise code. Once the driver has been chosen, a programmer has just to define the functions to be used at each step, without worrying about managing the underlying runtime. In fact, thanks to a high level of abstraction, ParSoDA-Py allows writing Python functions to execute the same functions in parallel on different runtimes (e.g., COMPS and Spark), simply changing the class of the driver in use (e.g., using the ParsodaPySparkDriver class for running the same application using PySpark).

5.3 Performance evaluation

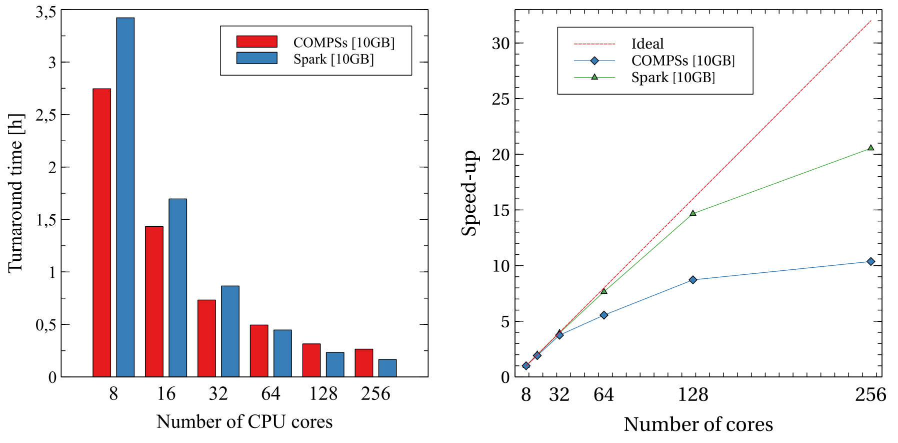
We carried out several experiments to evaluate and compare the performance of ParSoDA-Py on top of PyCOMPSs and PySpark, varying the dataset size and the number of cores used for execution. In particular, our specific focus was on assessing the two applications described using some datasets of social media posts with a different size (10 GB, 20 GB, 40 GB, and 180 GB).

The goal of the evaluation is to assess the parallel execution time and scalability of the ParSoDA-Py applications, obtained by varying the number of CPU cores exploited. In particular, the following performance parameters have been considered:

- *Turnaround time* the amount of time elapsed from the submission of an application to its end;
- *Speed-up* the ratio of the turnaround time using 1 worker node to the turnaround time using *n* worker nodes, which indicates how much performance gain is obtained by distributing data over an increasing number of nodes;
- *Scale-up* the turnaround time when the problem size is increased linearly with the number of CPU cores, which measures the capability of the system to manage increasing loads when computational resources are added to accommodate that growth.

5.3.1 Sequential pattern mining

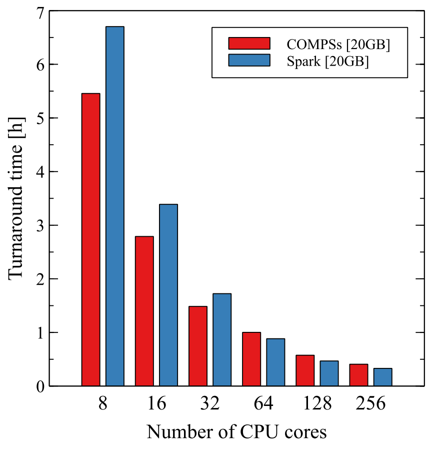
Figures 6, 7, 8 illustrate the turnaround time and relative speed-up of the sequential pattern mining application as the number of cores increases, considering different dataset sizes (10 GB, 20 GB, and 40 GB of geotagged posts). The results reported in the figures for the different dataset sizes show a turnaround time that



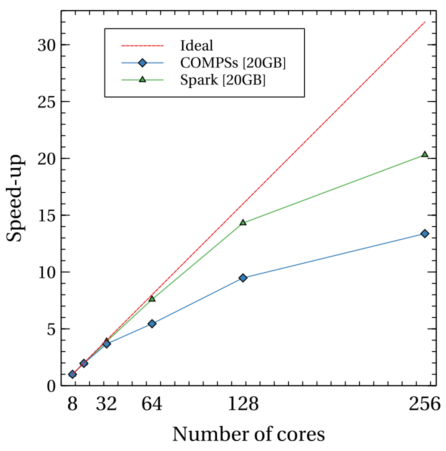
(a) Turnaround time with the 10GB dataset.

(b) Relative speed-up with the 10GB dataset.

Fig. 6 Trajectory mining application: execution time and speed-up of ParSoDA-Py using COMPSs and Spark, with a 10 GB dataset

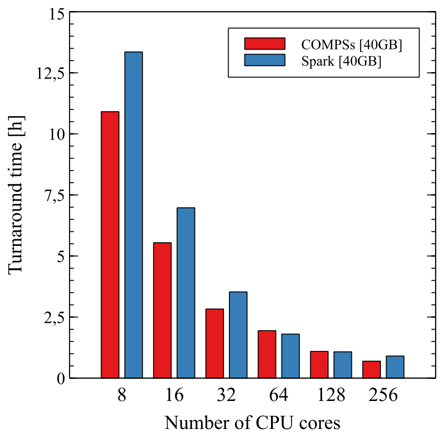


(a) Turnaround time with the 20GB dataset.

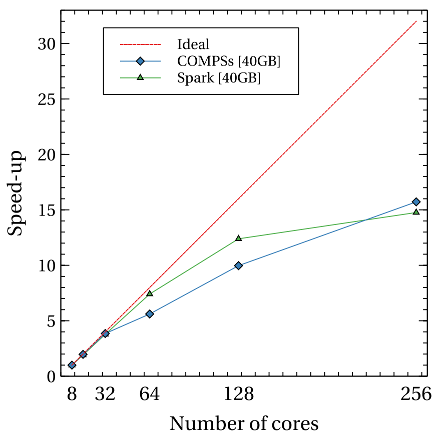


(b) Relative speed-up with the 20GB dataset.

Fig. 7 Trajectory mining application: execution time and speed-up of ParSoDA-Py using COMPSs and Spark, with a 20 GB dataset



(a) Turnaround time with the 40GB dataset.



(b) Relative speed-up with the 40GB dataset.

Fig. 8 Trajectory mining application: execution time and speed-up of ParSoDA-Py using COMPSs and Spark, with a 40 GB dataset

significantly decreases as the number of cores increases. In particular, Fig. 8a shows the execution time of the application when processing the 40 GB dataset. Using COMPSs, the turnaround time decreases from 10.9 h when using 8 CPU cores to 0.69 h when utilizing 256 CPU cores. Using Spark, the turnaround time is higher, in fact it decreases from 13.3 h when using 8 CPU cores to 0.9 h when utilizing 256 CPU cores. As illustrated in Fig. 8b, the speed-up using the 40 GB dataset is good, with values that are close to ideal up to 64 cores. More in detail, for COMPSs the speed-up is 1.97× on 16 cores, 3.85× on 20 cores, and 5.61× on 64 cores; for Spark, the speed-up is 1.91× on 16 cores, 3.78× on 32 cores, and 7.41× on 64 cores.

When comparing the performance of the two runtimes, COMPSs and Spark, it is worth noting that, on average, COMPSs outperformed Spark by 15.7% in terms of turnaround time. This performance improvement can influence the perceived speed-up values achieved by COMPSs, which may appear worse than those achieved by Spark. However, it's essential to consider that COMPSs' baseline (turnaround time with 8 cores) is significantly lower than Spark's baseline.

Figure 9 directly compares the scale-up of ParSoDA-Py on top of both COMPSs and Spark, showing the turnaround time obtained when the dataset size increases proportionally to the number of CPU cores used (i.e., from 10 GB using 8 cores, to 40 GB using 32 cores). This experiment is crucial for ensuring that the library can effectively handle an increase in data volume. In particular, the results shown in the figure highlight the system's ability to maintain constant execution times when the allocated cores increase proportionally to the size of the dataset. In particular, using the 10 GB with 8 cores, the turnaround time of the trajectory mining application is 2.74 h for COMPSs and 3.42 for Spark, for 20 GB with 16 cores is 2.79 h for COMPSs and 3.39 h for Spark, while for 40 GB with 32 cores is 2.8 h for COMPSs and 3.52 h for Spark. Overall, the results show that the system is able to manage the increasing computing load by increasing the number of processors.

5.4 Sentiment analysis

Figure 10 illustrates the execution time and relative speed-up of the sentiment analysis application, obtained with a dataset containing 180GB of posts. Also in this case, the results highlight a good reduction in the turnaround time as the number of cores increases, as the execution time decreases from 17.58 h on 8 CPU cores to 0.96 h on 256 CPU cores (Fig. 10a). Moreover, the speed-up is close to ideal values up to 64 cores (1.97× on 16 cores, 3.92× with 32 cores, and 6.26× on 64 cores). Also in this case, comparing the turnaround, COMPSs outperformed Spark by 20.6%. As discussed for the previous case study application, this performance difference influences the perceived speed-up of Spark (2.0× on 16 cores, 3.92× on 32 cores, and

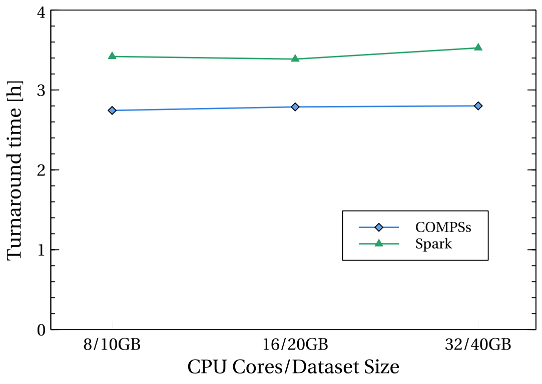
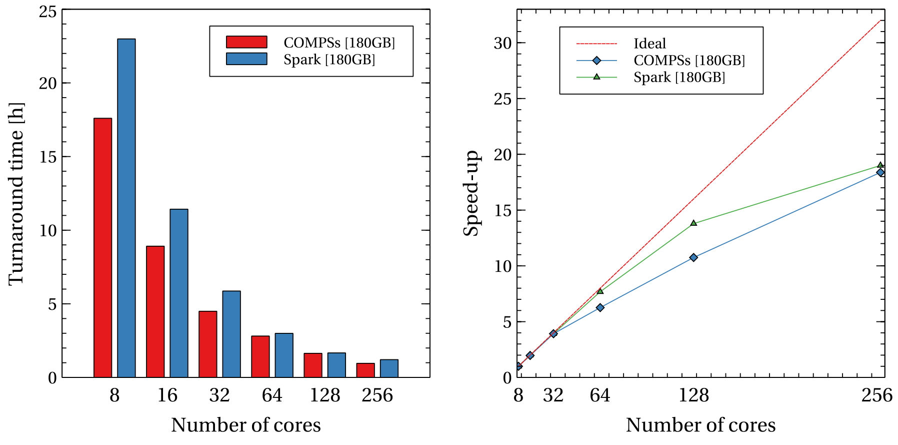


Fig. 9 Trajectory mining application: scale-up of ParSoDA-Py on top of COMPSs and Spark



(a) Turnaround time. (b) Relative speed-up.

Fig. 10 Sentiment analysis application: turnaround time and speed-up on top of COMPSs and Spark

7.68× on 64 cores), which apparently is better than that one achieved by COMPSs (Fig. 10b).

6 Conclusions

This paper focused on parallel data analysis and the challenges associated with the mining of large volumes of data, especially those coming from social media platforms. To address these challenges, the ParSoDA library was introduced as a powerful tool for building complex parallel social data analysis applications. Through the porting of ParSoDA in Python, namely ParSoDA-Py, we proved the versatility and usability of the library, leveraging on COMPSs and Spark to execute applications on different runtimes for HPC systems using Python. By considering new technical requirements, ParSoDA-Py represents a complete redesign of the original library, with extended support to other runtimes, data crawlers, and a vast and robust ecosystem of data processing libraries available in Python. The evaluation of ParSoDA-Py's usability and scalability through two real-world data analysis applications has provided valuable insights. The reduction in code complexity and the significant improvement in execution time on a private cluster with up to 256 cores emphasize the effectiveness of ParSoDA-Py in handling large-scale data analysis tasks. The availability of the ParSoDA-Py as open-source software further contributes to its accessibility and potential for wider adoption in the research and data mining communities. Researchers and developers can now leverage ParSoDA-Py's high-level programming structure and predefined functions to simplify the development of parallel data analysis applications. The ParSoDA-Py library is publicly available at https://github.com/SCAlabUnical/ParSoDA-Py. In addition, the library also gave us the opportunity to compare the performance of two HPC runtimes, COMPSs and

Spark, on the same applications. Experimental results show that COMPSs offers a more effective runtime support that is up to 20% faster than Spark.

Acknowledgements This work has been partially funded by the European Commission's Horizon 2020 Framework program and the European High-Performance Computing Joint Undertaking (JU) under Grant agreement No 955558 and by MCIN/AEI/10.13039/501100011033 and the European Union NextGenerationEU/PRTR (PCI2021-121957), project eFlows4HPC. It has also been supported by the Spanish Government (PID2019-107255GB) and by the Departament de Recerca i Universitats de la Generalitat de Catalunya to the Research Group MPiEDist (2021 SGR 00412) We also acknowledge financial support from "National Centre for HPC, Big Data and Quantum Computing," CN00000013 - CUP H23C22000360005, and from "FAIR - Future Artificial Intelligence Research" Project - CUP H23C22000860006.

Funding Open access funding provided by Università della Calabria within the CRUI-CARE Agreement.

References

1. Talia D, Trunfio P, Marozzo F (2015) Data analysis in the cloud: models. Techniques and Applications. Elsevier, Amsterdam, The Netherlands
2. Belcastro L, Marozzo F, Talia D, Trunfio P (2019) Parsoda: high-level parallel programming for social data mining. Soc Netw Anal Min 9(1):1
3. Belcastro L, Cantini R, Marozzo F, Orsino A, Talia D, Trunfio P (2022) Programming big data analysis: principles and solutions. J Big Data 9(4):1
4. Inoubli W, Aridhi S, Mezni H, Maddouri M, Mephu Nguifo E (2018) An experimental survey on big data frameworks. Futur Gener Comput Syst 86:546–564
5. Doulkeridis C, Vlachou A, Pelekis N, Theodoridis Y (2021) A survey on big data processing frameworks for mobility analytics. SIGMOD Rec 50(2):18–29
6. Talia D, Trunfio P, Marozzo F, Belcastro L, Cantini R, Orsino A (2024) Programming big data applications. World Scientific (Europe), Munich, Germany
7. Amer-Yahia S, Ibrahim N, Kengne CK, Ulliana F, Rousset M-C (2014) Socle: towards a framework for data preparation in social applications. Ingénierie des Systèmes d Inf 19(3):49–72
8. Cuesta Ã, Barrero DF, R-Doreno MD (2014) A framework for massive twitter data extraction and analysis. Malay J Comput Sci 27(1):50–67
9. Hussain A, Vatrapu R, Hardt D, Jaffari ZA (2014) Social data analytics tool: a demonstrative case study of methodology and software. In: Analyzing Social Media Data and Web Networks, pp. 99–118. Springer, Amsterdam, The Netherlands
10. Zhou D, Chen L, He Y (2015) An unsupervised framework of exploring events on twitter: Filtering, extraction and categorization. In: Proceedings of the AAAI Conference on Artificial Intelligence, vol. 29
11. You L, Motta G, Sacco D, Ma T (2014) Social data analysis framework in cloud and mobility analyzer for smarter cities. In: Proceedings of 2014 IEEE International Conference on Service Operations and Logistics, and Informatics, pp. 96–101 . IEEE
12. Elia D, Palazzo C, Fiore S, D'Anca A, Mariello A, Aloisio G (2023) Pyophidia: a python library for high performance data analytics at scale. SoftwareX 24:101538

13. Fiore S, Palazzo C, D'Anca A, Foster I, Williams DN, Aloisio G (2013) A big data analytics framework for scientific data management. In: 2013 IEEE International Conference on Big Data, pp. 1–8
14. Aldinucci M, Danelutto M, Kilpatrick P, Torquati M (2017) Fastflow: high-level and efficient streaming on multicore. Programming multi-core and many-core computing systems, 261–280
15. Löff J, Hoffmann RB, Pieper R, Griebler D, Fernandes LG (2022) Dsparlib: A c++ template library for distributed stream parallelism. Int J Parallel Prog 50(5–6):454–485
16. Rio Astorga D, Dolz MF, Fernández J, García JD (2017) A generic parallel pattern interface for stream and data processing. Concurr Comput: Pract Exp 29(24):4175
17. Belcastro L, Marozzo F, Talia D, Trunfio P (2017) A parallel library for social media analytics. In: The 2017 International Conference on High Performance Computing & Simulation (HPCS 2017), Genoa, Italy, pp. 683–690 . ISBN 978-1-5386-3250-5
18. Belcastro L, Marozzo F, Talia D, Trunfio P (2017) Appraising spark on large-scale social media analysis. In: Euro-Par Workshops. Lecture Notes in Computer Science, pp. 483–495, Santiago de Compostela, Spain . ISBN: 978-3-319-75178-8
19. Martin RC (1996) The open-closed principle. More C++ gems 19(96), 9
20. Tejedor E, Becerra Y, Alomar G, Queralt A, Badia RM, Torres J, Cortes T, Labarta J (2017) Pycompss: parallel computational workflows in python. IJHPCA 31(1):66–82
21. Mammadli N, Ejarque Artigas J, Álvarez Cid-Fuentes J, Badia Sala RM (2022) Dds: integrating data analytics transformations in task-based workflows [version 1; peer review: 1 approved, 2 approved with reservations]. Open Research Europe 2(article 66), 1–16
22. Belcastro L, Marozzo F, Perrella E (2021) Automatic detection of user trajectories from social media posts. Expert Syst Appl 186:115733
23. Li C et al. (2008) Efficiently mining closed subsequences with gap constraints
24. Chin D, Zappone A, Zhao J (2016) Analyzing twitter sentiment of the 2016 presidential candidates. Am J Sci Res
25. Kralj Novak P, Smailović J, Sluban B, Mozetič I (2015) Sentiment of emojis. PLOS ONE 10(12):1–22
26. Belcastro L, Cantini R, Marozzo F, Talia D, Trunfio P (2020) Learning political polarization on social media using neural networks. IEEE Access 8(1):47177–47187

Authors and Affiliations

Loris Belcastro[1,3] · **Salvatore Giampà**[3] · **Fabrizio Marozzo**[1,3] · **Domenico Talia**[1,3] · **Paolo Trunfio**[1,3] · **Rosa M. Badia**[2] · **Jorge Ejarque**[2] · **Nihad Mammadli**[2]

✉ Loris Belcastro
lbelcastro@dimes.unical.it

Salvatore Giampà
giampa@dtoklab.com

Fabrizio Marozzo
fmarozzo@dimes.unical.it

Domenico Talia
talia@dimes.unical.it

Paolo Trunfio
trunfio@dimes.unical.it

Rosa M. Badia
rosa.m.badia@bsc.es

Jorge Ejarque
jorge.ejarque@bsc.es

Nihad Mammadli
nihad.mammadli@bsc.es

[1] DIMES, University of Calabria, Rende, Italy

[2] Barcelona Supercomputing Center, Barcelona, Spain

[3] DtokLab Srl, Rende, Italy