

ParSoDA: High-Level Parallel Programming for Social Data Mining

Loris Belcastro · Fabrizio Marozzo ·
Domenico Talia · Paolo Trunfio

Received: date / Accepted: date

Abstract Software systems for social data mining provide algorithms and tools for extracting useful knowledge from user-generated social media data. ParSoDA (Parallel Social Data Analytics) is a high-level library for developing parallel data mining applications based on the extraction of useful knowledge from large dataset gathered from social media. The library aims at reducing the programming skills needed for implementing scalable social data analysis applications. To reach this goal, ParSoDA defines a general structure for a social data analysis application that includes a number of configurable steps and provides a predefined (but extensible) set of functions that can be used for each step. User applications based on the ParSoDA library can be run on both Apache Hadoop and Spark clusters. The paper describes the ParSoDA library and presents two social data analysis applications to assess its usability and scalability. Concerning usability, we compare the programming effort required for coding a social media application using versus not using the ParSoDA library. The comparison shows that ParSoDA leads to a drastic reduction (i.e., about 65%) of lines of code, since the programmer only has to implement the application logic without worrying about configuring the environment and related classes. About scalability, using a cluster with 300 cores and 1.2 TB of RAM, ParSoDA is able to reduce the execution time of such applications up to 85%, compared to a cluster with 25 cores and 100 GB of RAM.

Keywords Social Data analysis · Scalability · Parallel library · Big Data · Social media · Social networks

1 Introduction

Through the pervasive use of computers, smart phones and other digital objects, most human activities create big datasets whose collection, storage and analysis can be done. In particular, the use of social media produces a massive amount of data that can be downloaded from social media platforms or collected independently to understand human behaviors and processes. Social media mining aims at extracting useful knowledge from this big amount of data [3]. Social media analysis tools and algorithms have been used for the analysis of collective sentiments [16], for understanding the behavior of groups of people [6] or the dynamics of public opinion [15]. The use of parallel and distributed data analysis techniques and frameworks (e.g., MapReduce [10]) is essential to cope with the size and complexity of social media data. However, it is hard for many users to use such frameworks, mainly due to the programming skills needed for implementing the appropriate data analysis methods on top of them [18].

Loris Belcastro
DIMES Department, University of Calabria, Rende, Italy
E-mail: lbelcastro@dimes.unical.it

Fabrizio Marozzo · Domenico Talia · Paolo Trunfio
DIMES Department, University of Calabria, Rende, Italy
DtoK Lab Srl, Rende, Italy
E-mail: [fmarozzo, talia, trunfio]@dimes.unical.it

ParSoDA¹ (Parallel Social Data Analytics) is a programming library for simplifying the development of parallel social media mining application executed on High Performance Computing systems. To achieve this goal, ParSoDA provides a set of widely used functions for processing and analyzing data collected from social media, 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 social media analysis application that includes a number of steps (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.

In this way, data scientists and analysts having limited programming skills, especially with regard to parallel programming, can efficiently design and execute data analysis applications dealing with big amounts of social media data. The library includes algorithms that are widely used on social media data for extracting different kinds of information. To deal with social media items gathered from different sources, ParSoDA defines a metadata model that represents the different types of social media items (tweets, Flickr posts, etc.). The model can be easily extended to match most application requirements. Thus, the main contributions of ParSoDA are summarized as follows: *i*) ParSoDA defines a general structure for social data analysis applications and provides an extensible set of functions that can be used for each step; *ii*) ParSoDA aims at reducing the programming skills needed for implementing scalable social data analysis applications; *iii*) applications based on the ParSoDA library can be run on both Hadoop and Spark clusters; and *iv*) ParSoDA is able to reduce the execution time of data analysis by parallelizing the execution of the code and exploiting the computational and storage resources of clusters.

Parallel social data analysis applications based on the ParSoDA library can be run on Cloud and HPC systems exploiting both Apache Hadoop [4] and Spark [2]. To assess the usability and scalability of ParSoDA, we present two social data analysis applications that make use of the library to extract sequential patterns and frequent itemsets from social media data published in Flickr and Twitter. The first application aims at discovering sequential patterns from user movements, so as to find the common mobility patterns followed by users. The goal of the second application is to discover the frequent sets of places visited by users. Concerning usability, we compare the programming effort required for coding a social media application using versus not using the ParSoDA library. The comparison shows that ParSoDA leads to a drastic reduction (i.e., about 65%) of lines of code, since the programmer only has to implement the application logic without worrying about configuring the environment and related classes. The scalability was evaluated carrying out the data analysis applications on an HPC cluster. Using a cluster with 300 cores and 1.2 TB of RAM, ParSoDA is able to reduce the execution time of such applications up to 85%, compared to a cluster with 25 cores and 100 GB of RAM. This manuscript extends previous work [4] in the following main aspects: *i*) a formal description of the steps that compose a ParSoda application has been introduced; *ii*) the description and the pseudo code of the algorithms used in this paper for discovering behaviors and mobility patterns of users have been added; *iii*) a comparison about the programming effort required for coding a social media application using versus not using the ParSoDA library has been included; *iv*) more extensive and detailed scalability tests for measuring turnaround time, speed-up and scale-up have been added.

The remainder of the paper is organized as follows. Section 2 discusses related work. Section 3 describes the ParSoDA library. Section 4 presents the experimental evaluation of the two social data analysis applications. Finally, Section 5 concludes the paper.

2 Related work

Several developers and researches are working on the design and implementation of tools and algorithms for extracting useful information from data gathered from social media. In most cases the amount of data to be analyzed is so big that high-performance computers, such as many and multi-core systems, Clouds, and multi-clusters, paired with parallel and

¹ <https://github.com/SCA1abUnical/ParSoDA>

distributed algorithms, are used by data analysts to reduce response time to a reasonable value [3].

Several research projects consider not only the data analysis task, but also procedures including other 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.

SOCLE [1] is a framework for designing and optimizing data preparation in social applications. It is composed by a general-purpose three-layer architecture, an algebra, and a language for defining operations for data preparation in social applications. As an example, SOCLE provides operators to remove all unnecessary information from data (data pruning), to add information using external sources (data enrichment), and to transform data values (data normalization). The authors examined the use of SOCLE for manipulating social data in two families of social applications, recommendation and analytics, but no studies have been performed to assess its scalability, and no details about framework requirements have been provided.

Cuesta et al. [9] proposed a framework for easing Twitter data extraction and analysis. In the proposed architecture, the tweets, mined by the application through the Twitter APIs, are cleaned and then stored in a MongoDB database [7]. In addition to basic database operations (i.e., selection, projection, insertion, updating, and deletion), the framework can be extended creating more complex aggregation MapReduce tasks in Python. By default, the framework provides developers with modules for executing sentiment analysis and generating reports.

SODATO (SOcial Data Analytics Tool) [13] is an on-line tool for programming data analytics on social data. It utilizes the APIs provided by social media platforms (i.e., currently, it supports only Facebook and Twitter) for collecting data; then, it provides a combination of web as well as console applications that run in batches for pre-processing and aggregating data for analysis. At the end of the analytics process, the results can be displayed using the integrated visualization module. SODATO provides methods for several kinds of analysis, such as sentiments analysis, keyword analysis, content performance analysis, social influencer analysis, etc.

Zhou et al. [22] proposed a general unsupervised framework for exploring events from large amount of Twitter data. The framework exploits a pipeline process which consists of filtering, extraction, and categorization steps. During the filtering step, all event-related tweets are selected by exploiting a lexicon-based approach. Then, events are extracted from filtered tweets and grouped into categories using an unsupervised Bayesian model, called Latent Event & Category Model (LECM). The authors evaluated the categorization performances of the proposed framework on a dataset consisting of 60 million tweets, but no experiments on scalability have been provided.

Casalino et al. [5] presented a framework for exploring a collection of tweets by automatically extracting topics with semantic relevance (e.g. detect groups of tweets related to specific events and topics). The framework defines a process that consists of three steps. The first step transforms a collection of tweets according to a Vector Space Model. Then, during the second step, a Nonnegative Matrix Factorizations (NMF) technique is used to extract and cluster relevant topics from tweets. Finally, in the last step, a cluster analysis with word-cloud visualization is used to make a qualitative assessment of the results.

You et al. [20] presented a framework, running on Clouds, for developing social data analysis applications for smart cities, especially designed to support smart mobility. In particular, the framework is composed of five components (i.e., data collector, data preprocessor, data analyzer, data presenter, and data storage) that cover the whole data analysis life cycle. The framework supports data collection from social media platforms (e.g., Twitter, Foursquare), by exploiting their public APIs, and from other Internet sources (e.g., website, blog, files). A component devoted to data pre-processing provides functions for data cleansing, filtering, and normalization. Afterwards, the data analyzer component provides needed analysis methods (e.g., K-Means, DBSCAN, and Self-organizing Map) to make some data analysis.

The main differences between ParSoDA and the systems described above (but the one by You et al. [20]), are that our system was specifically designed to implement Cloud-based data analysis applications. To this end, 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 The ParSoDA library

ParSoDA (Parallel Social Data Analytics) is a Java library that includes algorithms widely used to process and analyze data gathered from social media with the goal of extracting different kinds of information (e.g., user mobility, user sentiments, topic trends, and frequency).

ParSoDA defines a general structure for a social data analysis application that is composed of the steps described in the following.

Data acquisition The goal of this step is to collect social media items from a set of data sources $DS = \{ds_0, \dots, ds_m\}$. Given a data source ds_i , a query q_i is defined such that $q_i(ds_i) = \{g_0, \dots, g_n\}$, where g_j is a social media item gathered from ds_i by applying q_i . Given DS and an ordered set of queries $Q = \{q_0, \dots, q_m\}$, a data acquisition function can be defined as $\mathbf{af}(DS, Q) = \{q_0(ds_0) \cup \dots \cup q_m(ds_m)\}$.

Data filtering This step filters the social media items gathered in the previous step, based on a set of functions. A filtering function $\mathbf{ff}(g_i) = \{true \vee false\}$ is a boolean function that takes as input a social media item g_i and verifies whether g_i matches some criteria. Given a set of filtering functions $\mathbf{FF} = \{\mathbf{ff}_0, \dots, \mathbf{ff}_m\}$ and a set of social media items $G = \{g_0, \dots, g_n\}$, $\mathbf{FF}(G) = \{g_i \in G \mid (\mathbf{ff}_0(g_i) \wedge \dots \wedge \mathbf{ff}_m(g_i)) = true\}$.

Data mapping This step modifies the filtered social media items by applying a set of mapping functions. A mapping function $\mathbf{mf}(g_i) = g'_i$ where g'_i is a modified social media item derived from g_i . Given a set of mapping functions $\mathbf{MF} = \{\mathbf{mf}_0, \dots, \mathbf{mf}_m\}$ and a set of social media items $G = \{g_0, \dots, g_n\}$, $\mathbf{MF}(G) = \{g'_i \mid g'_i = \mathbf{mf}_m(\mathbf{mf}_{m-1}(\dots \mathbf{mf}_0(g_i)))\}$.

Data partitioning This step partitions a set of social media items into shards using a group key, and then sorts all the items with the same group key according to a sort key. A function $\mathbf{gk}(g_i)$ returns the group key of g_i , while a function $\mathbf{sk}(g_i)$ returns the sort key of g_i . Given a key k and a set of social media items $G = \{g_0, \dots, g_n\}$, all the items having k as group key are $G_k = \{g_{k_i} \in G \mid \mathbf{gk}(g_{k_i}) = k\}$. Then, a set $G_k = \{g_{k_0}, \dots, g_{k_m}\}$ can be ordered according to \mathbf{sk} by generating an ordered set of items $(G'_k) = \{g_{k_i} \in G_k \mid \mathbf{sk}(g_{k_0}) \leq \dots \leq \mathbf{sk}(g_{k_m})\}$. Given a group key function \mathbf{gk} , a sort key function \mathbf{sk} , and all possible group keys $K = \{k_0, \dots, k_p\}$, a partitioning function is defined as $\mathbf{pf}(G, \mathbf{gk}, \mathbf{sk}) = \{G'_{k_0}, \dots, G'_{k_p}\}$.

Data reduction This step aggregates all the data into a shard according to a reduce function, that generates a new set of elements $E = \{e_0, \dots, e_n\}$. Given a set of items (G'_k) associated to a key k , a reduce function is defined as $\mathbf{rf}(G'_k) = E$.

Data analysis This step analyzes the outputs of the data reduction step by using a data mining function to extract a set of patterns $P = \{p_0, \dots, p_n\}$. Given a set of elements $E = \{e_0, \dots, e_m\}$, a data mining function is defined as $\mathbf{af}(E) = P$.

Data visualization This is the final step in which a visualization function is applied to the data analysis results to generate a set of visual results $R = \{r_0, \dots, r_n\}$. Given a set of patterns $P = \{p_0, \dots, p_m\}$, a visualization function is defined as $\mathbf{vf}(P) = R$.

For each of these steps, ParSoDA provides a predefined set of functions. For example, for the data acquisition step, ParSoDA provides crawling functions for gathering data from some of the most popular social media platforms (e.g., 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. Users are free to extend this set of functions with their own. The current version of the library (v. 1.3.0 dated October 25, 2018) contains more than forty predefined functions organized in seven packages, corresponding to the seven ParSoDA steps. Details on each function are available at <https://github.com/SCALabUnical/ParSoDA>.

3.1 Reference architecture and execution flow

Figure 1 presents the reference architecture and execution flow of a ParSoDA application that runs on the Hadoop [19] or Spark [21] framework. In such way, it is possible to implement several parallel and distributed data mining applications with high scalability [8]. As shown in Figure 1(a), user applications can utilize ParSoDA and other libraries (e.g., Mahout², MLlib³). Applications can be executed on Hadoop or Spark, using YARN as resource manager and HDFS as distributed storage system. Figure 1(b) provides details on how applications are executed on a Hadoop or a Spark cluster. The cluster is formed by one or more master nodes, and multiple worker nodes. Once a user application is submitted to the cluster, its steps are executed according to their order (i.e., data acquisition, data filtering, etc.).

On a Hadoop cluster, some steps are inherently MapReduce-based, namely: *data filtering*, *data mapping*, *data partitioning* and *data reduction*. This means that all the functions used to perform these steps are executed within a MapReduce job that runs on a set of worker nodes. In particular, the data filtering and data mapping steps are wrapped within Hadoop Map tasks; the data partitioning step corresponds to Hadoop Split and Sort tasks; the data reduction step is executed as a Hadoop Reduce task. The remaining steps (data acquisition, data analysis, and data visualization) are not necessarily MapReduce-based. This means that the functions associated with these steps could be executed in parallel on multiple worker nodes, or alternatively they could be executed locally by the master node(s). The latter case does not imply that execution is sequential, because a master node can make use of some other parallel runtime (e.g., MPI).

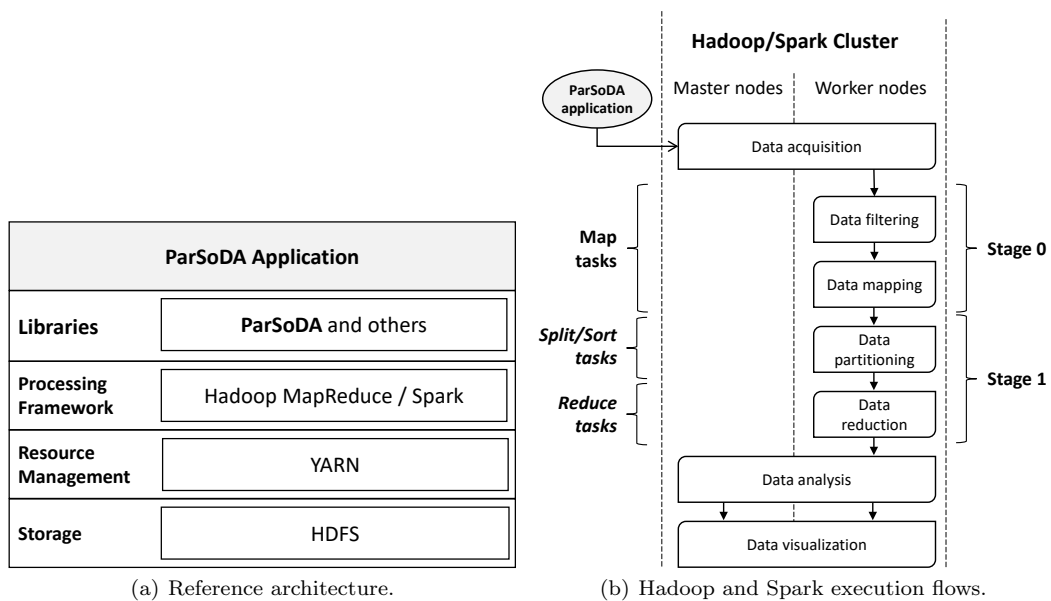


Fig. 1 Reference architecture and execution flow.

On a Spark cluster, 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), the same considerations made for Hadoop apply to Spark.

² <https://mahout.apache.org/>

³ <https://spark.apache.org/mllib/>

3.2 Metadata model for social media data

To deal with social media items gathered from different sources, ParSoDA defines a metadata model for representing the different types of social media items (tweets, Flickr posts, etc.). According to this model, each social media item is represented by a metadata document composed of two parts: a *basic* section that includes fields common to all social media platforms (source, item id, date and time, location coordinates, user info); an *extra* section that contains fields specific to the source. As an example, Listing 1 shows a metadata element describing a tweet. The *source* field indicates that it is a social media item gathered from Twitter, and therefore the *extra* section contains fields specific to the tweets (whether it is a retweet or not, the retweet count, and so on). Listing 2 contains a metadata element for a Flickr photo. The *source* field indicates that it is a Flickr social media item, and thus the *extra* section contains fields specific to Flickr photos (a list of tags, date when the photo was taken and so on).

```
{
  "BASIC":{
    "SOURCE":"Twitter", "ID":"111222333444555",
    "DATETIME":"2015-12-20T23:20:34.000",
    "LOCATION":{"LNG":-0.1262,"LAT":51.5011},
    "USER":{"USERID":"12345", "USERNAME":"joedoe"}},
  "EXTRA":{
    "inReplyToScreenName":"billsmith", "inReplyToUserId":123456789,
    "inReplyToStatusId":678712345678962848,
    "text":"@billsmith that sounds great!",
    "hashtags":["#code", "#mapreduce"], "retweets":0, "isRetweet":false}
}
```

Listing 1 Metadata of a tweet serialized in JSON format.

```
{
  "BASIC":{
    "SOURCE":"Flickr", "ID":"43012793876",
    "DATETIME":"2016-11-21T22:12:36.000",
    "LOCATION":{"LNG":12.456661,"LAT":41.90245},
    "USER":{"USERID":"111222333@N00", "USERNAME":"mrwho"}},
  "EXTRA":{
    "title":"Basilica di San Pietro",
    "description":"St Peter's church in Rome"
    "tags":[{"count":0,"value":"holiday"}, {"count":0,"value":"vatican"}],
    "dateTaken":"Nov 9, 2016 12:00:00 AM",
    "accuracy": 16}
}
```

Listing 2 Metadata of a Flickr photo serialized in JSON format.

ParSoDA defines an abstract class named *SocialItem* that defines the *basic* fields, and a set of classes (*TwitterSocialItem*, *FlickrSocialItem*, etc.) that extend *SocialItem* by defining the *extra* fields specific to different social media. Each social media item is represented in memory by an instance of one such classes (e.g., a tweet will be an instance of *TwitterSocialItem*). When the metadata of a social media item must be saved to persistent storage or sent through the network, the object is serialized in JSON format, a widely-used text notation [11].

3.3 Structure of a ParSoDA application

ParSoDA defines a general structure for a social data analysis application that includes a number of steps (data acquisition, filtering, mapping, partitioning, reduction, analysis, and visualization), and provides a predefined (but extensible) set of functions for each step. Thus, an application developed with ParSoDA is expressed by a concise code that specifies the functions invoked at each step. More specifically, a ParSoDA application can be developed by creating an instance of a class named *SocialDataApplication*, which defines a set of methods that allow the programmer specifying the functions to be used at each step.

Table 1 lists the main methods of the *SocialDataApplication* class. For each method, the table specifies the step which it refers to, and a short description.

Table 1 Main methods of the *SocialDataApplication* class.

Step	Function and description
Data acquisition	<i>setCrawlers(Class[] functions, String[] params)</i> Specifies the crawling functions to be used for data acquisition. The <i>functions</i> array contains the crawling classes; <i>params[i]</i> contains the configuration string of <i>functions[i]</i> .
Data filtering	<i>setFilters(Class[] functions, String[] params)</i> Specifies the functions and associated parameters to be used to perform data filtering.
Data mapping	<i>setMapFunctions(Class[] functions, String[] params)</i> Specifies the functions and associated parameters to be applied at the mapping step.
Data partitioning	<i>setPartitioningKeys(String groupKey, String sortKey)</i> Specifies the keys used by the <i>secondary sort</i> pattern, which partitions data into shards by a primary key (<i>groupKey</i>) and then sorts shards by a secondary key (<i>sortKey</i>).
Data reduction	<i>setReduceFunction(Class function, String params)</i> Specifies the function and associated parameters to be used at the reduction step.
Data analysis	<i>setAnalysisFunction(Class function, String params)</i> Specifies the function and associated parameters to be used to perform data analysis.
Data visualization	<i>setVisualizationFunction(Class function, String params)</i> Specifies the function and associated parameters to be used for data visualization.

For the *Data acquisition* step, the *SocialDataApplication* class provides the *setCrawlers* method that can be used to specify which crawling functions will be used to collect data from social media platforms. The method receives two arrays as parameters: *functions* and *params*. Array *functions* contains the crawling classes that will be instantiated to perform data collection. Array *params* contains the parameters that are necessary to configure the instances of the crawling classes specified in *functions*; specifically, *params[i]* contains the configuration string of *functions[i]*. In ParSoDA, a set of crawling classes are available. For example, a *FlickrCrawler* class can be instantiated to collect data from the Flickr social network. If *functions* specifies multiple crawling classes, they will be instantiated and executed in parallel.

Data filtering is configured with the *setFilters* method. It works similarly to the *setCrawlers* method. In fact, its first parameter can be used to specify the names of the classes that will be instantiated to perform data filtering, while the second one contains the parameters used to configure the instances of the filtering classes. ParSoDA implements a simple but effective mechanism for filtering social media items according to a set of conditions. Each filtering class implements a predicate function that verifies if a social media item meets or not a particular condition. In ParSoDA, a filtering class is defined by implementing the interface *Predicate*, which is included in Java since version 1.8. After executing the data filtering step, only social media items that match all conditions provided will be passed to the data mapping step.

For configuring the *Data mapping* step, the developer has to use the *setMapFunctions* method. Similarly to the methods described above, it receives two arrays as parameters, which specify, respectively, the names of the classes that will be instantiated to perform data mapping and the parameters used to configure them. In ParSoDA a mapping class defines a function that transforms a filtered social media item. In such way, developers can transform social media items by applying a sequence of map functions. A map function is defined by extending the abstract class *MapFunction*.

Data partitioning is configured with the *setPartitioningKeys* method, which receives two strings as parameters: *groupKey* and *sortKey*. The method partitions data in shards by *groupKey* and then sorts all data in a shard by *sortKey*. The keys used to configure this step must be present in the metadata model used to represent the social media items under processing. ParSoDA implements the *Secondary Sort* design pattern [19], which allows configuring a primary key (*groupKey*) for partitioning data into shards, and a secondary key (*sortKey*) for sorting all data in a shard. As an example, this mechanism can be used to

partition data by user ids and then sort it by timestamps, which is a very common task in sequential pattern mining.

Data reduction is configured with the *setReduceFunction* that has two input parameters: the name of the class that will be instantiated to perform data reduction and the parameters used to configure it. The reduce function aggregates all the data contained in a shard. As an example, to analyze the movements of social media users, one might use a reduce function for aggregating all the data of a single user according to given criteria. In ParSoDA, a reduce function is defined by creating a class that implements the interface *ReduceFunction*.

Data analysis is configured with the *setAnalysisFunction* method, which receives as input parameters the name of the class that will be instantiated to perform the data analysis task and the associated parameters. A data analysis function is defined by extending the abstract class *AnalysisFunction*, which requires the implementation of two abstract methods: *formatData*, for formatting the input data in the format required by the analysis function, and *analyzeData* that implements the data analysis algorithm.

Finally, *Data visualization* is configured with the *setVisualizationFunction* method that, similar to the previous methods, receives the name of the data visualization class and the parameters required to create an instance. To create a custom data visualization function, the programmer must define a class that implements the interface *VisualizationFunction*.

4 Case study applications

We carried out a set of experiments to evaluate usability and scalability of ParSoDA on two social data analysis applications that process data published in Flickr and Twitter. The first application aims at discovering sequential patterns from user movements, so as to find the common routes followed by users. The second one aims at discovering the frequent sets of places visited by users. The analysis was carried out by analyzing 325 GB of social media data published in Flickr and Twitter that refer to the center of Rome.

The remainder of this section is organized as follows. Section 4.1 shows the code of the ParSoDA applications. Section 4.2 describes the pseudo code of the sequential pattern analysis and frequent itemset analysis algorithms used in the applications. Section 4.3 discusses the most interesting results that have been obtained by analyzing the dataset. Section 4.4 assesses the usability of ParSoDA by comparing the programming effort required for coding data analysis applications using versus not using the ParSoDA library. Finally, Section 4.5 presents scalability tests for measuring turnaround time, speed-up and scale-up.

4.1 Application code

Listing 3 shows the code of the application for executing the sequential pattern analysis. First, an instance of the *SocialDataApp* class must be created (*line 3*). Then a file containing the boundaries of the regions of interest (*RomeRoIs.kml*) is distributed to the processing nodes (*lines 4-5*).

```

1 public class SequentialPatternMain {
2     public static void main(String[] args) {
3         SocialDataApp app = new SocialDataApp("SPM - City of Rome");
4         String[] cFiles = {"RomeRoIs.kml"};
5         app.setDistributedCacheFiles(cFiles);
6         Class[] cFunctions = {FlickrCrawler.class, TwitterCrawler.class};
7         String[] cParams = {"-lng 12.492 -lat 41.890 -radius 10 -startDate 2014-11-01 -
            endDate 2016-07-31", "-lat 12.492 -lng 41.890 -radius 10 -startDate 2014-11-01
            -endDate 2016-07-31"};
8         app.setCrawlers(cFunctions, cParams);
9         Class[] fFunctions = {IsGeotagged.class, IsInPlace.class};
10        String[] fParams = {"", "-lat 12.492 -lng 41.890 -radius 10"};
11        app.setFilters(fFunctions, fParams);
12        Class[] mFunctions = {FindPoI.class};
13        String[] mParams = null;
14        app.setMapFunctions(mFunctions, mParams);
15        String groupKey = "USER.USERID";

```



```

16 String sortKey = "DATETIME";
17 app.setPartitioningKeys(groupKey,sortKey);
18 Class rFunction = ReduceByTrajectories.class;
19 String rParams = "-t 5";
20 app.setReduceFunction(rFunction,rParams);
21 Class aFunction = PrefixSpan.class;
22 String aParams = "-maxPatternLength 5 -minSupport 0.005";
23 app.setAnalysisFunction(aFunction,aParams);
24 Class vFunction = SortPrefixSpanBy.class;
25 String vParams = "-k support -d DESC";
26 app.setVisualizationFunction(vFunction,vParams);
27 app.execute();
28 }
29 }

```

Listing 3 A sequential pattern analysis application written using ParSoDA.

Afterwards, the different steps of the application are configured as described in the following:

1. *Data acquisition.* The names of two crawling classes (*FlickrCrawler* and *TwitterCrawler*) are defined in the *cFunctions* array (line 6). The parameters used to configure the instances of the two crawling classes are defined in the *cParams* array (line 7). The two arrays are then passed to the *setCrawlers* method (line 8).
2. *Data filtering.* Two filtering classes are specified: *IsGeotagged* and *IsInPlace* (line 9). The former filters data by keeping only geotagged items. The latter filters out data that are not in the center of Rome, which is defined by its geographical coordinates. The parameters of the two filtering functions are specified in the *fParams* array (line 10). The names of the filtering classes and associated parameters are then passed to the *setFilters* method (line 11).
3. *Data mapping.* The map class *FindPoI* (line 12), which does not require parameters to be instantiated (line 13), is specified. The mapping function defined in *FindPoI* assigns to each social media item the name of the place that it refers to. To do this, it refers to the boundaries specified in the file defined at line 4. The name of the map class is then passed to the *setMapFunctions* method (line 14).
4. *Data partitioning.* The id of the user who posted a social media item is used as the *groupKey* (line 15), while the date and time when the social media item was posted is used as the *sortKey* (line 16). The two keys are then passed to the *setPartitioningKeys* method (line 17).
5. *Data reduction.* A reduce class, named *ReduceByTrajectories* (line 18), is specified to aggregate all the social media items posted by a single user, into a list of individual trajectories across places. The parameters of the reduce class are specified in the *rParams* string (line 19). In particular, it receives only a parameter *t*, which is the maximum time gap in hours that can be taken for consecutive places in the same trajectory. The name of the reduce class and its parameters are then passed to the *setReduceFunction* method (line 20).
6. *Data analysis.* A data analysis class, named *PrefixSpan*, is specified (line 21). The class implements PrefixSpan [17], a scalable frequent sequence mining algorithm, which takes as input a collection of sequences and mines frequent sequences. The parameters of data analysis class are specified in the *aParams* string (line 22). The name of the data analysis class and its parameters are then passed to the *setAnalysisFunction* method (line 23). More details about this algorithm are in Section 4.2.
7. *Data visualization.* The *SortPrefixSpanBy* class is specified to perform the data visualization function (line 24). A configuration string *vParams*, containing the parameters of the data visualization class, is specified at line 25. The class receives two parameters: the key used to sort results (the sequence support) and the sort direction (descending order). The name of the data visualization class and its parameters are then passed to the *setVisualizationFunction* method (line 26).

The sequential pattern analysis application is executed by invoking the *execute* method (line 27).

The frequent itemset analysis application is slightly different from the sequential pattern analysis application. In Listing 4 we show only the differences with Listing 3, which are detailed in the following:

- a different reduce class, named *ReduceByItemsets*, is specified to aggregate all the social media items into a set of individual places posted by each user (*line 18*). Similarly to the sequential pattern analysis application, it receives only a parameter t , which is the maximum time gap in hours for consecutive places in the same trajectory (*line 19*).
- for extracting frequent sets of places from social media data, a parallel implementation of FP-Growth [12], called PFP [14], has been used (*lines 21-23*). This algorithm receives a parameter *minSupport*, which represents the minimal support level of the frequent pattern.

```

17 ...
18 Class rFunction = ReduceByItemsets.class;
19 String rParams = "-t 5";
20 app.setReduceFunction(rFunction,rParams);
21 Class aFunction = FPGrowth.class;
22 String aParams = "-minSupport 0.005";
23 app.setAnalysisFunction(aFunction,aParams);
24 ...

```

Listing 4 An excerpt of the ParSoDa code for implementing a frequent itemset analysis.

4.2 Data analysis algorithms

Data is analyzed for discovering behaviors and mobility patterns of users. Specifically, we perform both *sequential pattern analysis* and *frequent item sets analysis*, as described in the following. The input dataset S is a list of individual sequences (or trajectories) across places obtained after the reduction step. Formally,

$$S = \{s_0, s_2, \dots, s_n\}$$

where the i -th sequence $s_i = \langle u_i, \{(p_{i0}, t_{i0}), \dots, (p_{im}, t_{im})\} \rangle$, where (p_{ij}, t_{ij}) is a pair indicating *place* and *timestamp* of the j -th post published by user u_i .

4.2.1 Sequential Pattern Analysis.

Sequential pattern analysis algorithms are intended to discover the sequences of elements that occur most frequently in the data. Unlike the frequent item set analysis, in sequential analysis are fundamental the time dimension and the chronological order in which the values appear. In our case, this type of analysis is useful to discover the most frequent mobility patterns across places. In our experiments we used a sequential pattern mining function that is based on the PrefixSpan algorithm [17] (see Algorithm 1).

The function receives as input a dataset S containing sequences of places and the minimum support sup_{min} . Given S , the function creates a list data structure R for storing the frequent sequential patterns, an empty *prefix*, and sets to 1 the starting length for the sequential pattern L (lines 1-4). Then, the *PrefixSpan* sub-function is called (line 5). This sub-function makes a set of recursive calls, until all the frequent patterns are found. Finally, the function returns the list of frequent sequential patterns of places grouped by pattern length (line 6). The *PrefixSpan* sub-function receives as input a dataset S of sequential patterns, the *prefix* of the sequential patterns, the length of the patterns L , the minimum support sup_{min} , and the list R where to store the patterns found. If the given dataset is empty, the sub-function ends (line 8). If not, it scans S to find L_1 , i.e., the list of all frequent sequential patterns of length 1 in S (line 10). Each element of L_1 is a pair $\langle pattern, count \rangle$, representing the pattern and its associated support count. For each element $l_i \in L_1$, the algorithm iterates (lines 11-15) performing the following operations:

- Discards all the patterns having a support count lower than the minimum support sup_{min} (line 12).

ALGORITHM 1: Sequential Pattern Mining algorithm pseudo code.

```

Input : A dataset containing sequences of places  $S$ , minimum support  $sup_{min}$ 
Output: Frequent sequential patterns of places  $R$ 
1 Function SequentialPatternMining( $S, sup_{min}$ )
2    $R \leftarrow []$ ; /* List for storing frequent sequential patterns */
3    $prefix \leftarrow \vdash$ ; /* Variable to store temporary prefix */
4    $L \leftarrow 1$ ; /* Initial pattern length */
5   PrefixSpan( $prefix, L, S, sup_{min}, R$ ); /* Recursive call */
6   return  $R$ 
7 SubFunction PrefixSpan( $prefix, L, S, sup_{min}, R$ )
8   if  $S == \emptyset$  then
9     return;
10   $L_1 = \text{findLength1SequentialPatterns}(S)$ ;
11  foreach  $l_i \in L_1$  do
12    if  $l_i.count \geq sup_{min}$  then
13       $R[L].\text{add}(\langle prefix + l_i.pattern, l_i.count \rangle)$ ; /* Add to results */
14       $SB_i = \{ \text{suffixes of trajectories in } S \text{ having } l_i \text{ as prefix} \}$ 
15      PrefixSpan( $prefix + l_i.pattern, L + 1, SB_i, sup_{min}, R$ ); /* Recursion */
16  return;

```

- If the support count of l_i is greater than or equal to the minimum support, then stores a new frequent pattern of length L , obtained by concatenating the pattern of l_i to the current prefix (line 13).
- Calculates a new dataset of sequences SB_i containing all the suffixes of the sequential patterns in S having l_i as prefix (line 14).
- Calls recursively the *PrefixSpan* sub-function to calculate on SB_i sequential frequent patterns of length $L + 1$ for the currently calculated prefix (line 15).

In our experiments, a parallel version of *PrefixSpan*, which is included in the Spark Machine Learning library (MLlib), has been used. For large datasets, the computations of length-1 sequential patterns F (line 10) and of the database SB_i (line 14) are executed in parallel by splitting the original dataset into shards that are processed locally. At the end, the final results are obtained by merging the local frequent patterns.

4.2.2 Frequent Itemset Analysis.

Frequent itemset analysis is exploited with the goal of discovering the items that occur together with a high frequency. Applied to dataset S , we perform a frequent place sets discovery task, aimed at extracting the sets of places that are most frequently visited together by users. In our experiments we used the FPGrowth algorithm [12] whose pseudo-code is shown in Algorithm 2.

ALGORITHM 2: FPGrowth pseudo code.

```

Input : A dataset containing sequences of places  $S$ , minimum support  $sup_{min}$ 
Output: Frequent sets of places  $FS$ 
1  $F \leftarrow []$ ; /* List of frequent items */
2  $I \leftarrow \{\}$ ; /* Set of items in  $S$  */
3 foreach  $s_i \in S$  do
4   foreach  $l_i \in s_i$  do
5      $F[l_i] ++$ ;
6      $I \leftarrow I \cup \{l_i\}$ ;
7 SortAndRemove( $F, sup_{min}$ ); /* Sort F and remove infrequent items */
8  $root \leftarrow \vdash$ ; /* Create a null FP-tree root */
9 foreach  $s_i \in S$  do
10  SortByFrequency( $s_i, F$ ); /* Sort  $s_i$  according to  $F$  */
11  BuildTree( $s_i, root, sup_{min}$ ); /* Insert items in  $s_i$  in the FP-Tree */
12 foreach  $i \in I$  do
13  Growth( $root, i, sup_{min}, FS$ ); /* Call the recursive method Growth() */
14 return  $FS$ 

```

The algorithm receives a dataset S containing the sequences of places and the minimum support sup_{min} . It initially calculates a list F , containing the supports of each place in the dataset, and a places' vocabulary I , containing the unique places found by scanning S (lines 1-6). Then, the items in F are sorted by support and infrequent entries (i.e., places with a support lower than sup_{min}) are removed (line 7). Next, the algorithm scans all the trajectories in S (line 9-11) to build a tree, called *FP-Tree* [17], where each path on the tree represents a frequent set sharing the same places. According to the information contained in F , each sequence in $s_i \in S$ is sorted by descending order of their frequency in the dataset (line 10). Then, it inserts the sorted sequence in the FP-Tree structure by inserting only places in s_i that meet minimum support threshold (line 11). Since places in each sequences have to be sorted by descending order of their frequency in the dataset, the tree can be processed quickly. Finally, for each unique place found in the dataset, the recursive processing of the FP-Tree is executed for finding all frequent item sets. (line 13). Finally, all the frequent itemsets found are returned (line 14).

We used a parallel implementation of FPGrowth, PFP[14], which is able to run in parallel all the steps of FPGrowth. Specifically, the computations of frequent places in F (lines 1-6) is parallelized by sharding and parallel counting. Moreover, the recursive calls to tree building operation and the *Growth* procedure (lines 9-13) are parallelized by splitting the original dataset into shards and then by aggregating the results.

4.3 Application results

A set of 24 popular places in the center of Rome have been considered to run the sequential pattern mining task and the frequent itemset discovery task, both implemented as ParSoDA applications. In the following, we discuss some of the most interesting results that have been obtained by analyzing the real dataset.

The sequential pattern analysis has been carried out for discovering the most frequent routes in Rome. In this experiment, it has been set a maximum time duration (gap) to move from a place to another of 5 hours. This means that, if the time distance between two contiguous places in sequence is greater than 5 hours, they will belong to different sequences. Table 2 provides detailed information about the most frequent patterns and the corresponding supports.

Table 2 Top 5 sequential patterns of length 2, 3, and 4 across places in Rome

Sequential pattern	Support
Colosseum → St. Peter's Basilica	9.07%
St. Peter's Basilica → Colosseum	7.72%
Colosseum → Roman Forum	5.28%
Colosseum → Pantheon	4.44%
Colosseum → Trevi Fountain	4.19%
Colosseum → Roman Forum → St. Peter's Basilica	4.4%
Vatican Museums → St. Peter's Basilica → Colosseum	3.9%
Colosseum → Trevi Fountain → St. Peter's Basilica	3.7%
Colosseum → Roman Forum → Pantheon	3.6%
Colosseum → Pantheon → St. Peter's Basilica	3.6%
Colosseum → Trevi Fountain → Pantheon → St. Peter's Basilica	0.64%
Colosseum → Roman Forum → Trevi Fountain → San St. Peter's Basilica	0.61%
Colosseum → Roman Forum → Piazza Venezia → Piazza di Spagna	0.58%
Colosseum → Roman Forum → Piazza Venezia → St. Peter's Basilica	0.58%
Colosseum → Roman Forum → Pantheon → St. Peter's Basilica	0.58%

Considering the sequential patterns of length 2, the sequence $\{Colosseum \rightarrow St. Peter's Basilica\}$ is the most frequent route among places in Rome, followed by 9.07% of users. The sequence $\{Colosseum \rightarrow Roman Forum \rightarrow St. Peter's Basilica\}$ is the most frequent route of length 3, which is followed by 4.4% of users. Finally, the sequence $\{Colosseum \rightarrow Trevi Fountain \rightarrow Pantheon \rightarrow St. Peter's Basilica\}$ is the most frequent route of length 4 with a quite low support of 0.64%. As an example, Figure 2 shows the top five interesting patterns of length 3 which have been found by the PrefixSpan algorithm.

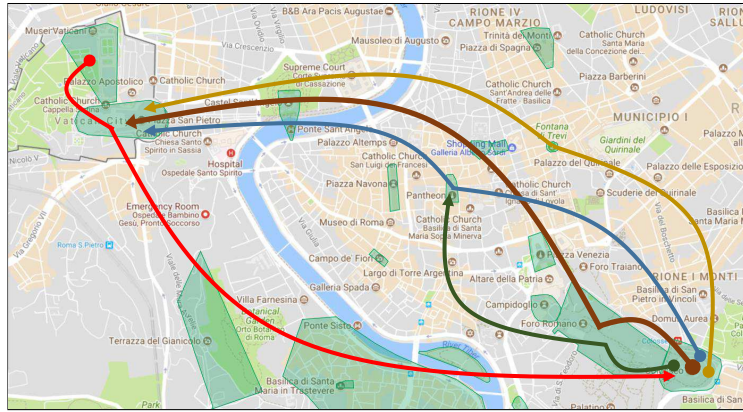


Fig. 2 Top 5 sequential patterns of length 3.

Table 3 shows the top 5 places visited in Rome, with the corresponding support in the data. The Colosseum is the most visited place, followed by the St. Peter's Basilica.

Table 3 Top 5 places visited in Rome

Place	Support
Colosseum	21.7%
St Peter's Basilica	13.9%
Trastevere	8.7%
Pantheon	6.5%
Trevi Fountain	5.3%

Table 4 Top 5 frequent sets of places visited in Rome

Set of places	Support
Pantheon, St. Peter's Basilica, Colosseum	5.3%
Trevi Fountain, St. Peter's Basilica, Colosseum	4.5%
Roman Forum, St. Peter's Basilica, Colosseum	4.4%
Vatican Museums, St. Peter's Basilica, Colosseum	4.4%
Trevi Fountain, Pantheon, Colosseum	4.0%

Table 4 shows the most frequent itemsets of length 3 that have been discovered by the PFP algorithm. Set $\{Pantheon, St. Peter's Basilica, Colosseum\}$ is the most frequent set of places visited by social users in Rome, with a support of 5.3%. Combining the information contained in Tables 3 and 4, an interesting result is that *Trastevere*, a popular district of Rome, is the third most visited place, but it is not present in any frequent itemset. This could happen because Trastevere is visited by people during the evening, for having a dinner in one of its many restaurants or pubs, but it is not part of common tourist routes during the daylight.

4.4 Usability evaluation

Writing a parallel data analysis application from scratch usually requires deep programming skills and the writing of many lines of code. In fact, designing and implementing such kind of applications pose a number of challenges to developers such as parallelization of complex algorithms, reduction of communication costs, and optimization of memory usage. In this section, we demonstrate the usability of ParSoDA by comparing the programming effort required for coding data analysis applications using versus not using the ParSoDA library. In particular, in the first part of the section, we discuss how the sequential pattern analysis application shown in Listing 3 could be re-implemented in Hadoop without ParSoDA. In the second part, we point out the reduction of lines of code obtained using ParSoDA.

Listing 5 shows the main program of the sequential pattern analysis application implemented in Hadoop without ParSoDA (a similar code is obtained by coding the application in Spark). A developer has to design the entire MapReduce job by implementing several classes (e.g., input/output formats, partitioner, group and sort comparator). The whole process is not easy to do and requires a deep knowledge of the Hadoop/Spark architecture.

```

1 public class ApplicationDriverHadoop {
    public static class TextPair implements WritableComparable<TextPair> {
        ...
    }
36 }

```

```

37 private static Configuration conf = new Configuration();
38 private static FileSystem fs = null;
39 private static Job job = null;
40 public static void main(String[] args) throws Exception {
41     String pathFlickrItems = "FlickrRome2017.json";
42     String pathRoIs = "RomeRealShapes.kml";
43     String outputBasePath = "outputMR/";
44     conf = new Configuration();
45     conf.set("fs.defaultFS", "file:///");
46     fs = FileSystem.get(conf);
47     job = Job.getInstance(conf);
48     job.setJobName("Extracting user movements from Rome Flickr dataset");
49     job.addCacheFile(new Path(pathRoIs).toUri());
50     MultipleInputs.addInputPath(job, new Path(pathFlickrItems), TextInputFormat.
        class, DataMapper.class);
51     TextOutputFormat.setOutputPath(job, new Path(outputBasePath+"dataset"));
52     job.setMapOutputKeyClass(MainMR.TextPair.class);
53     job.setMapOutputValueClass(Text.class);
54     job.setPartitionerClass(SecondarySort.SSPartitioner.class);
55     job.setGroupingComparatorClass(SecondarySort.SSGroupComparator.class);
56     job.setSortComparatorClass(SecondarySort.SSSortComparator.class);
57     job.setReducerClass(DataReducer.class);
58     job.setNumReduceTasks(1);
59     job.setOutputKeyClass(NullWritable.class);
60     job.setOutputValueClass(Text.class);
61     fs.delete(new Path(outputBasePath), true);
62     boolean jobSuccessful = job.waitForCompletion(true);
63     String params = "-i " + outputBasePath+"dataset/part-r-00000 -o " +
        outputBasePath + "mgfsm" + " -m d -g 3 --tempDir /tmp";
64     try {
65         String[] analysisParams = params.split(" ");
66         ToolRunner.run(new Configuration(), new FsmDriver(), analysisParams);
67     } catch (Exception e) { e.printStackTrace(); }
68     String vis_params = outputBasePath + "mgfsm/translatedFS/part-r-00000 " +
        outputBasePath + "sortedFS";
69     VisualizeFSMResult.visualize(vis_params.split(" "));
70 }
71 }

```

Listing 5 The main program of the sequential pattern analysis application implemented in Hadoop without ParSoDA.

In addition to the main program, other MapReduce classes must be implemented to code the functions required by the application. For instance, mining data from different social media requires writing many lines of code to implement crawlers for collecting data. Moreover, to deal with data coming from different social media, specific parsers are needed for converting data into a common format. To address this issues, ParSoDA includes some crawlers and parsers and defines a common metadata model for representing data coming from different social media. Without ParSoDA, for implementing data filtering and mapping functions into a MapReduce application, a specific mapper class must be defined. Listing 6 shows the Hadoop mapper class that implements the data filtering functions (*isGeotagged*, *isInPlace*) and mapping function (*FindPoI*) used in Listing 3. The mapper is quite complex and requires many lines of code. In contrast, ParSoDA allows developers to use predefined functions, or to implement new ones by coding only the function logic, without worrying about configuring the Hadoop environment and related classes.

```

1 public class DataMapper extends Mapper<LongWritable, Text, TextPair, Text> {
    ...
11 protected void setup(Mapper<LongWritable, Text, TextPair, Text>.Context context)
        throws IOException, InterruptedException {
12     this.context = context;
13     loadRoIs();

```

```

14 List<Predicate<GeotaggedItem>> filterFunctions = new LinkedList<Predicate<
    GeotaggedItem>>();
15 filterFunctions.add(new IsGeotagged());
16 filterFunctions.add(new IsInRome());
17 itemFilter = filterFunctions.stream().reduce(e -> true, Predicate::and);
18 super.setup(context);
19 }
20 public void map(LongWritable key, Text value, Context context) throws
    InterruptedException, IOException {
21     item = buildItem(value.toString());
22     if (item == null)
23         return;
24     if (itemFilter != null && !itemFilter.test(item))
25         return;
26     item = assignLocation(item, rois);
27     if (item == null)
28         return;
29     outputKey.left = new Text(item.search(mapperGroupKey).toString());
30     outputKey.right = new Text(item.search(mapperGroupSortKey).toString());
31     outputValue.set(item.toString());
32     context.write(outputKey, outputValue);
33 }
34 private static GeotaggedItem buildItem(String s) {
    ...
40 }
41 private static GeotaggedItem assignLocation(GeotaggedItem g, List<Roi> rois) {
    ...
57 }
58 private void loadRois() throws IOException {
    ...
87 }
    ...
98 }

```

Listing 6 Mapper class of the sequential pattern analysis application implemented in Hadoop.

With regards to the data partition step, it is necessary to implement the secondary sort pattern, which enables to partition and sort data passed to each reducer. To do that, some efforts and skills on the functioning of the Hadoop partition phase are required. Specifically, four classes (*SSPartitioner*, *SSGroupComparator*, *SSSortComparator*, *TextPair*) have to be implemented, which require to write more than 120 lines of code. Instead, the secondary sort pattern is a built-in functionality of ParSoDA, which can be easily configured from the application main.

Similarly, for implementing the data reduction function, a specific reducer class must be defined. Listing 7 shows the Hadoop reducer class that implements the data reduction function for extracting user trajectories across locations (*ReduceByTrajectories* in Listing 3). In ParSoDA, user data can be easily aggregated in trajectories using a predefined function.

For the data analysis and visualization steps, ParSoDA provides a set of predefined functions ready to use or alternatively an abstract class that can be extended to recall existent algorithms available in external libraries or packages. Using an abstract class allows developers to easily integrate algorithms within a social media application.

```

1 public class ReduceByTrajectories extends Reducer<TextPair, Text, NullWritable,
    Text> {
    ...
11 @Override
12 protected void setup(Reducer<TextPair,Text,NullWritable,Text>.Context context)
    throws IOException,InterruptedException {
    ...
18 }
19 @Override
20 public void reduce(TextPair key, Iterable<Text> values, Context context) throws
    java.io.IOException, InterruptedException {

```

```

21 List<String> res = concatenateLocationsByDay(values);
22 for (String tmp : res) {
23     outputValue.set(tmp);
24     context.write(NullWritable.get(), outputValue);
25 }
26 }
27 private List<String> concatenateLocationsByDay(Iterable<Text> listItems, int
    dayStep) {
28     LocalDateTime oldTimestamp = new LocalDateTime(0);
29     LocalDateTime currTimestamp = null;
30     ret = new LinkedList<String>();
31     oldLocation = null;
32     currentLocation = null;
33     item = null;
34     s = null;
35     for (Text value : listItems) {
36         ...
60     }
61     return ret;
62 }
63 }

```

Listing 7 Reducer class of the sequential pattern analysis application implemented in Hadoop.

In general, using ParSoDA leads to a drastic reduction of lines of code. As reported in Table 5, it allows to achieve an overall reduction of 65% of lines of code for implementing the sequential pattern analysis application. In particular, ParSoDA allows programmers to save hundred lines of code in the main (as the programmer needs to specify only the functions to be used and their parameters), in the data acquisition and data partition steps (where built-in functionalities are exploited), as well as in the data filtering, mapping, and reduction steps (where the programmer needs only to define the function logic). For the data analysis and visualization steps, we used the same code to invoke external libraries, which does not lead to a gain in terms of lines of code. However, for these steps, ParSoDA ensures many advantages in terms of usability. In fact, in the application main defined through ParSoDA, all the MapReduce jobs created for the different steps, such as the ones in the analysis and visualization steps, are automatically chained. This means that the output of a job is automatically used as input to the next step. In contrast, without ParSoDA, programmers need to manually control the execution flow among different jobs.

Table 5 Total number of lines of code of the sequential pattern analysis application using versus not using ParSoDA.

Step	ParSoDA	Hadoop
Main	29	71
Data acquisition	0	220
Data filtering	30	40
Data mapping	26	98
Data partitioning	0	120
Data reduction	5	63
Data analysis	120	120
Data visualization	75	75
Total	285	803

4.5 Scalability evaluation

We experimentally evaluated the scalability of ParSoDA by running the data analysis applications on a private cloud infrastructure. Specifically, we used a cluster equipped with 1 head node and 12 worker nodes, each one with 25 CPU cores and 100 GB of memory (altogether there are 300 CPU cores and 1200 GB of memory). In our experiments, we used

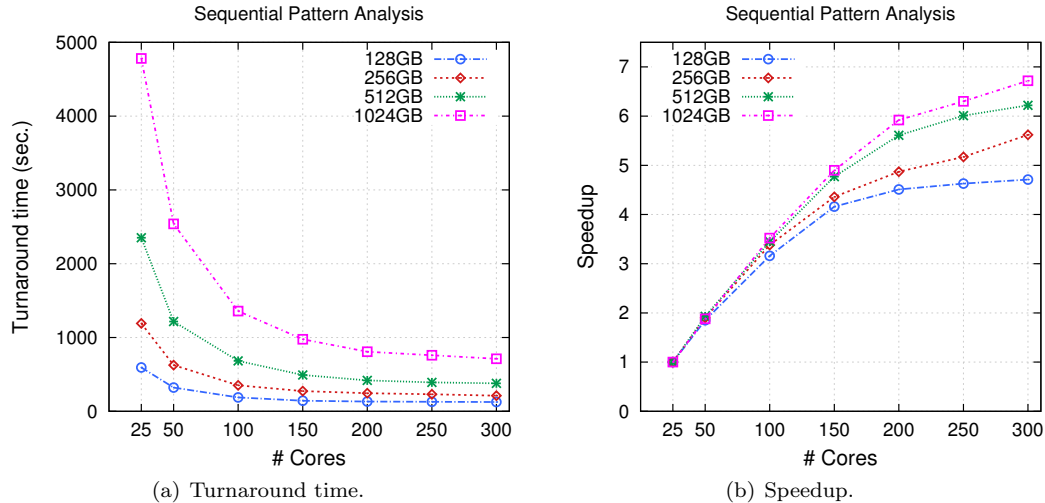


Fig. 3 Turnaround time and relative speedup of the sequential pattern analysis for different data sizes.

the Spark version of ParSoDA, since, as demonstrated in [2], it resulted to be faster than the Hadoop version of the library.

The analysis was carried out by analyzing a dataset containing 325 GB of social media items published by Flickr and Twitter users from 2014 to 2016 referring to the center of Rome. To perform a more complete scalability analysis, we randomly sampled the original dataset to generate four datasets D_{128} , D_{256} , D_{512} , D_{1024} that contains 128 GB, 256 GB, 512 GB and 1024 GB of data respectively.

The goal of the evaluation is to assess the scalability of the ParSoDA applications, obtained by varying the number of worker nodes used (i.e., the number of 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 worker nodes, which measures the capability of the system to manage increasing loads when machines are added to accommodate that growth.

Figure 3 shows the results obtained with the sequential pattern analysis application. Figure 3(a) shows the turnaround times of the ParSoDA application for the four datasets considered using from 25 to 300 CPU cores (i.e., from 1 to 12 worker nodes). For the smallest dataset (D_{128}) the turnaround time decreases from 10 minutes using 25 cores to 2 minutes using 300 cores. For D_{256} the turnaround time decreases from 20 to 3.5 minutes. For D_{512} the turnaround time decreases from 39 to 6.3 minutes. Finally, for the largest dataset (D_{1024}), the turnaround time ranges from 1.3 hour to 12 minutes. The scalability achieved using ParSoDA can be evaluated through Figure 3(b), which illustrates the relative speedup obtained by using up to 300 cores. For the smallest dataset (D_{128}) the speedup passes from 1.9 using 50 cores to 4.7 using 300 cores. For D_{256} the speedup ranges from 1.9 to 5.6. For D_{512} the speedup ranges from 1.9 to 6.2. Finally, for the largest dataset (D_{1024}), we obtained a speedup ranging from 1.9 to about 7. As shown in Figure 3(b), although the speedup is not ideal, as the size of the analyzed dataset increases, also the system speedup increases, because the system is able to exploit the parallel processing capability.

Figure 4 shows the results obtained with the frequent itemset analysis application. As shown, the turnaround time (Figure 4(a)) and speedup (Figure 4(b)) are similar to those obtained with the sequential pattern analysis application. However, some differences can be found by analyzing separately the contribution of step 6 (data analysis) from that of steps 1-5 (pre-processing), as described in Figure 5.

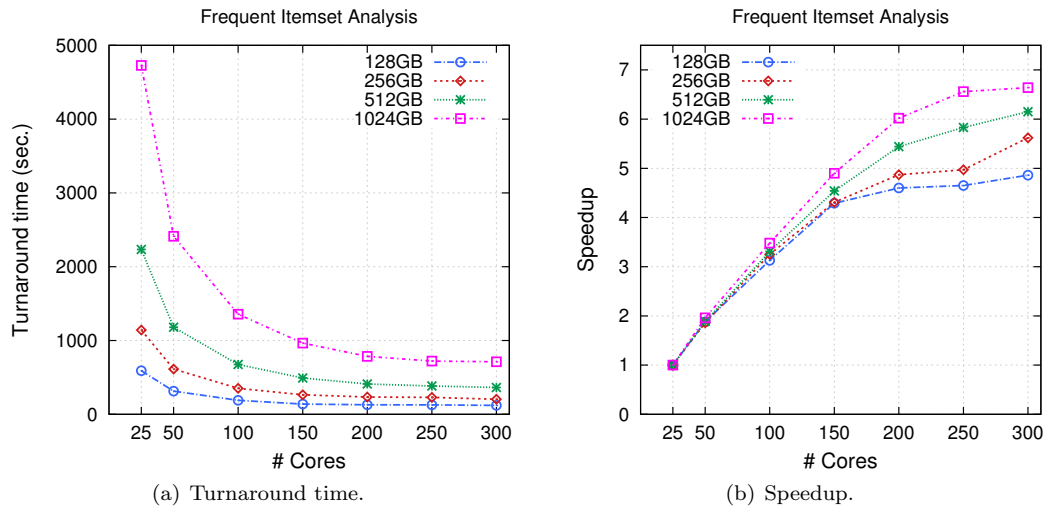


Fig. 4 Turnaround time and relative speedup of the frequent itemset analysis for different data sizes.

Figure 5 measures the scale-up of both applications by showing the turnaround time obtained when the dataset size increases proportionally to the number of worker nodes used (i.e., from 128 GB using 25 cores, to 1024 GB using 200 cores). As mentioned before, the figure also shows the time required by the ParSoDA applications for executing pre-processing (steps 1-5) and data analysis (step 6).

Using the *D128* with 25 cores the turnaround time of the sequential pattern analysis application is 9.9 minutes, for *D256* with 50 cores is 10.4 minutes, for *D512* with 100 cores is 11.4 minutes, while for *D1024* with 200 cores is 13.4 minutes. Instead, the turnaround time of the frequent itemset analysis application is 9.8 minutes, for *D256* with 50 cores is 10.2 minutes, for *D512* with 100 cores is 11.2 minutes, while for *D1024* with 200 cores is 13 minutes.

In both cases, the results show that the turnaround time increases moderately as the data size and the number of cores become greater. In particular, by analyzing the partial times, we observe that the data analysis step increases slightly more than the other steps of the application. Overall, the results show that the system is able to manage the increasing computing load by increasing the number of processors.

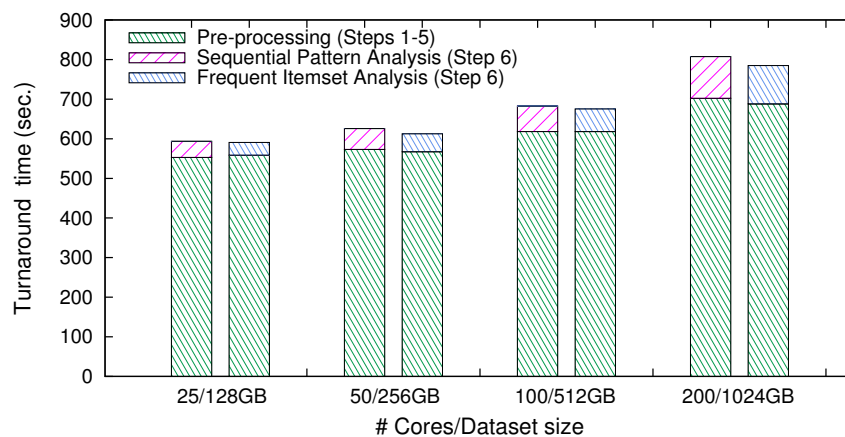


Fig. 5 Scale-up: Turnaround time vs number of cores/dataset size.

5 Conclusions

Social data mining is an important research area aimed at extracting useful information from the big amount of data gathered from social media. To cope with the size and complexity of social media data, the use of parallel and distributed data mining techniques is fundamental. ParSoDA is a high-level library that can be used for building complex parallel social data analysis applications. It defines a general structure for social data analysis programming that includes a number of key steps (data acquisition, filtering, mapping, partitioning, reduction, analysis, and visualization), and provides a predefined (but extensible) set of functions for each step.

Parallel social data analysis applications based on the ParSoDA library can be run on Cloud and HPC systems exploiting both Apache Hadoop and Spark. To assess the usability and scalability of ParSoDA, we discussed two social data analysis applications implemented through the library to extract sequential patterns and frequent itemsets from social media data published in Flickr and Twitter. About usability, ParSoDA leads to a drastic reduction (i.e., about 65%) of lines of code, since the programmer only has to implement the application logic without worrying about configuring the environment and related classes. About the scalability, the performance has been evaluated by running the data analysis applications on a private Spark cluster with 300 cores and 1.2 TB of RAM. The results demonstrate that ParSoDA is able to reach a good level of scalability as it allowed to reduce the execution time up to 85%, compared to the execution on a cluster with 25 cores and 100 GB of RAM.

The ParSoDA library is available as open-source software at <https://github.com/SCALabUnical/ParSoDA>.

Acknowledgment

This work has been partially supported by the SMART Project, CUP J28C17000150006, funded by Regione Calabria (POR FESR-FSE 2014-2020) and by the ASPIDE Project funded by the European Unions Horizon 2020 research and innovation programme under grant agreement No 801091.

References

1. Amer-Yahia, S., Ibrahim, N., Kengne, C.K., Ulliana, F., Rousset, M.C.: Socle: Towards a framework for data preparation in social applications. *Ingénierie des Systèmes d'Information* **19**(3), 49–72 (2014)
2. Belcastro, L., Marozzo, F., Talia, D., Trunfio, P.: Appraising spark on large-scale social media analysis. In: *Euro-Par Workshops, Lecture Notes in Computer Science*, pp. 483–495. Santiago de Compostela, Spain (2017). ISBN: 978-3-319-75178-8
3. Belcastro, L., Marozzo, F., Talia, D., Trunfio, P.: Big data analysis on clouds. In: S. Sakr, A. Zomaya (eds.) *Handbook of Big Data Technologies*, pp. 101–142. Springer (2017). ISBN: 978-3-319-49339-8
4. Belcastro, L., Marozzo, F., Talia, D., Trunfio, P.: A parallel library for social media analytics. In: *The 2017 International Conference on High Performance Computing & Simulation (HPCS 2017)*. Genoa, Italy (2017)
5. Casalino, G., Castiello, C., Buono, N.D., Mencar, C.: A framework for intelligent twitter data analysis with non-negative matrix factorization. *IJWIS* **14**(3), 334–356 (2018)
6. Cesario, E., Iannazzo, A.R., Marozzo, F., Morello, F., Riotta, G., Spada, A., Talia, D., Trunfio, P.: Analyzing Social Media Data to Discover Mobility Patterns at EXPO 2015: Methodology and Results. In: *The 2016 International Conference on High Performance Computing and Simulation (HPCS 2016)*. Innsbruck, Austria (2016)
7. Chodorow, K.: *MongoDB: the definitive guide*. " O'Reilly Media, Inc." (2013)
8. Chu, C., Kim, S.K., Lin, Y.A., Yu, Y., Bradski, G., Ng, A.Y., Olukotun, K.: Map-reduce for machine learning on multicore. *Advances in neural information processing systems* **19**, 281 (2007)
9. Cuesta, Á., Barrero, D.F., R-Moreno, M.D.: A Framework for massive Twitter data extraction and analysis. *Malaysian J. of Computer Science* **27**, 1 (2014)
10. Dean, J., Ghemawat, S.: Mapreduce: Simplified data processing on large clusters. In: *Proceedings of the 6th Conference on Symposium on Operating Systems Design & Implementation, OSDI'04*, pp. 10–10. Berkeley, USA (2004)
11. ECMA: Ecma-262: ECMAscript Language Specification. Fifth edition. ECMA (European Association for Standardizing Information and Communication Systems) (2009)
12. Han, J., Pei, J., Yin, Y., Mao, R.: Mining frequent patterns without candidate generation: A frequent-pattern tree approach. *Data mining and Knowledge Discovery* **8**(1), 53–87 (2004)
13. Hussain, A., Vatrupu, R.: *Social Data Analytics Tool (SODATO)*, pp. 368–372. Springer International Publishing, Cham (2014)

14. Li, H., Wang, Y., Zhang, D., Zhang, M., Chang, E.Y.: Pfp: Parallel fp-growth for query recommendation. In: Proceedings of the 2008 ACM Conference on Recommender Systems, pp. 107–114. New York, NY, USA (2008)
15. Marozzo, F., Bessi, A.: Analyzing Polarization of Social Media Users and News Sites during Political Campaigns. *Social Network Analysis and Mining* **8**(1) (2018)
16. Pang, B., Lee, L.: Opinion mining and sentiment analysis. *Foundations and Trends in Information Retrieval* **2**(12), 1–135 (2008)
17. Pei, J., Han, J., Mortazavi-Asl, B., Wang, J., Pinto, H., Chen, Q., Dayal, U., Hsu, M.C.: Mining sequential patterns by pattern-growth: the prefixspan approach. *IEEE Transactions on Knowledge and Data Engineering* **16**(11), 1424–1440 (2004)
18. Talia, D., Trunfio, P., Marozzo, F.: *Data Analysis in the Cloud*. Elsevier (2015)
19. White, T.: *Hadoop: The definitive guide*. ” O’Reilly Media, Inc.” (2012)
20. You, L., Motta, G., Sacco, D., Ma, T.: 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 (2014)
21. Zaharia, M., Xin, R.S., Wendell, P., Das, T., Armbrust, M., Dave, A., Meng, X., Rosen, J., Venkataraman, S., Franklin, M.J., et al.: Apache spark: A unified engine for big data processing. *Communications of the ACM* **59**(11), 56–65 (2016)
22. Zhou, D., Chen, L., He, Y.: An unsupervised framework of exploring events on twitter: Filtering, extraction and categorization. In: Proceedings of the Twenty-Ninth AAAI Conference on Artificial Intelligence, January 25-30, 2015, Austin, Texas, USA., pp. 2468–2475 (2015)