A High-Level Programming Library for Mining Social Media on HPC Systems

Loris BELCASTRO [a], Fabrizio MAROZZO [a,b,1], Domenico TALIA [a,b], and Paolo TRUNFIO [a,b]
e-mail: [lbelcastro, fmarozzo, talia, trunfio]@dimes.unical.it
[a] *DIMES Department, University of Calabria, Rende, Italy*
[b] *DtoK Lab Srl, Rende, Italy*

Abstract. The convergence between HPC and Big Data processing can be pursued also providing high-level parallel programming tools for developing Big data analysis. 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 data mining applications on HPC systems 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 goal of this paper is to assess the *flexibility* and *usability* of the ParSoDA library. Through some code snippets, we demonstrate how programmers can easily extend ParSoDA functions on their own if they need any custom behavior. Concerning the usability, we compare the programming effort required for coding a social media application using or 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.

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 [1]. Social media analysis tools and algorithms have been used for the analysis of collective sentiments [2], for understanding the behavior of groups

[1] Corresponding Author; E-mail: fmarozzo@dimes.unical.it

of people [3] or the dynamics of public opinion [4]. The convergence between HPC and Big Data processing can be pursued also providing high-level parallel programming tools for developing Big data analysis. Software systems for social data mining provide algorithms and tools for extracting useful knowledge from user-generated social media data. The use of HPC systems and parallel and distributed data analysis techniques and frameworks (e.g., MapReduce [5]) 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 [6].

ParSoDA[2] (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 on clouds and HPC systems. 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.

Parallel social data analysis applications based on the ParSoDA library can be run on Cloud and HPC systems exploiting both Apache Hadoop [7] and Spark [8]. To demonstrate the benefits of ParSoDA in terms of flexibility and usability, we present a social data analysis application that make use of the library to extract sequential patterns from social media data published in Flickr and Twitter. Through some code snippets, we demonstrate how programmers can easily extend ParSoDA functions on their own if they need any custom behavior. Concerning the usability, we compare the programming effort required for coding a social media application using and not using the ParSoDA library. The comparison shows that ParSoDA leads to a reduction of 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 remainder of the paper is organized as follows. Section 2 discusses related work. Section 3 describes the ParSoDA library. Section 4 presents a social data analysis application for extracting sequential patterns and examines the benefits of using ParSoDA over coding the same application without using any high-level programming library. Finally, Section 5 concludes the paper.

[2] `https://github.com/SCAlabUnical/ParSoDA`

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 distributed algorithms, are used by data analysts to reduce response time to a reasonable value [1].

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 [9] is a framework for designing and optimizing data preparation in social applications. It is composed by a general-purpose three-layers 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 by using external sources (data enrichment), 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. [10] 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 [11]. 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) [12] 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. [13] 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 by 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. [14] 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. [15] 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. [15]), is 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 that are widely used to process and analyze data gathered from social media for extracting different kinds of information (e.g., user mobility, user sentiments, topic trends).

ParSoDA defines a general structure for a social data analysis application that is formed by the following steps:

- *Data acquisition*: during this step, it is possible to run multiple crawlers in parallel; the collected social media items are stored on a distributed file system (HDFS [16]).
- *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 is 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.

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 some of the most popular social media (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.

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 it refers to, and a short description. More details on these methods can be found in [17].

Table 1. Main methods of the *SocialDataApplication* class.

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

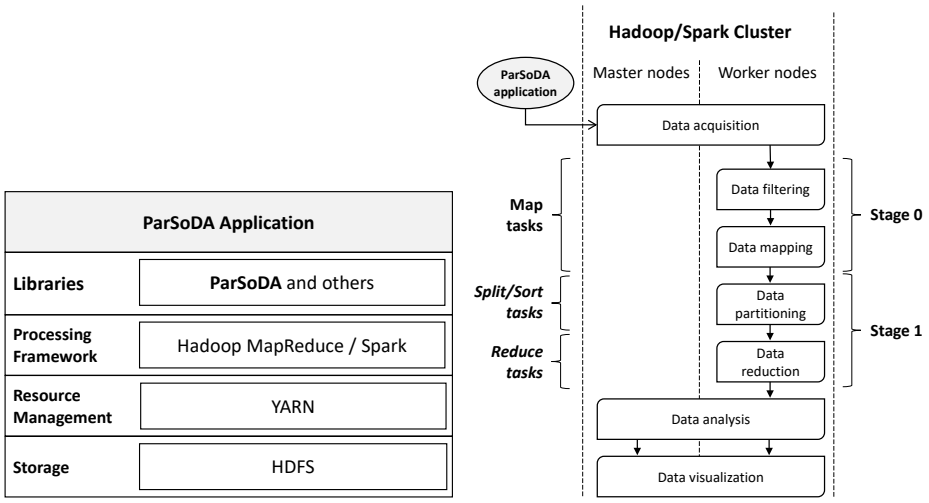
Figure 1 presents the reference architecture and execution flow of a ParSoDA application that runs on the Hadoop [18] or Spark [19] framework. In such way, it is possible to implement several parallel and distributed data mining applications with high scalability [20]. As shown in the Figure 1(a), user applications can utilize ParSoDA and other libraries (e.g., Mahout[3], MLlib[4]). Applications can be executed on Hadoop or Spark,

[3]https://mahout.apache.org/

[4]https://spark.apache.org/mllib/

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 to 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).



(a) Reference architecture. (b) Hadoop and Spark execution flows.

Figure 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.

3.1. 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":"Jan 1, 0001 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 [21].

4. Case study application

We evaluated the flexibility and usability of ParSoDA through a social media application to discover sequential patterns from user movements. 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. Moreover, we performed a code comparison between the application written using ParSoDA and the same application coded without using any high-level programming library.

4.1. Application code using ParSoDA

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 [22], 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*).
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*).

An example of the results obtained by the application is shown in Figure 2, which illustrates the top five interesting patterns of length 3 that have been found by the PrefixSpan algorithm. More details on the results of this application are discussed in [7].

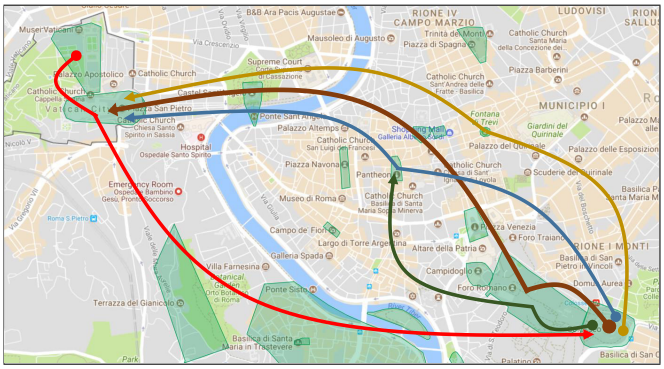


Figure 2. Top 5 sequential patterns of length 3.

4.2. Extending the ParSoDA library

ParSoDA provides a predefined set of functions for each step described in the previous section, such as:

- *data acquisition*, ParSoDA provides crawling functions to collect data from the most popular social media (Twitter and Flickr);
- *data filtering*, it provides functions for filtering geotagged items based on their position, time of publication, and contained keywords;
- *data reduction*, it provides utility classes for aggregating social media items according to a parametric property.

However users are free to extend these functions with their own if they need any custom behavior. In the following, we present some code snippets on how to extend the main ParSoDA classes.

As an example, for the *data acquisition* step, the predefined *FlickrCrawler* class can be used for collecting data from Flickr. To create a custom crawler, developers must extend the *AbstractCrawlerFunction* class and define the abstract method *collect*.

```
1 public abstract class AbstractCrawlerFunction {
2  Object[] params;
3  public AbstractCrawlerFunction(Object[] params) {
4   this.params = params;
5  }
6  public abstract void collect();
7  ...
8 }
```

Listing 4: Abstract class for implementing a crawling function in ParSoDA.

For the *data filtering* step, developers have to: *i*) create a class extending the abstract class *AbstractFilterFunction*, and *ii*) define the *test* method by implementing its own logic. The method receive one parameter that is a generic *GeotaggedItem*, so as it is able to process social media items coming from different social media platforms. Listing 5 shows a simple function used to filter out social media items that are not geotagged.

```
public class IsGeotagged extends AbstractFilterFunction {
 public boolean test(GeotaggedItem g) {
   return (g.getLocation()!= null)? true : false;
 }
}
```

Listing 5: Example of a filtering function using ParSoDA.

For the *data mapping* step, a new function can be created by extending the *AbstractMapFunction* class (see Listing 6) and implementing its behavior in the abstract method *apply*, which can be used to transform, add, or remove the information contained in each geotagged item. For example in our application, we used a simple map function that loads a set of points-of-interest from a distributed file and assigns to each social media item the name of the place it refers to. As shown in Listing 7, this map function can be implemented by writing few lines of code and without modifying other classes. It is also possible to chain multiple map functions (i.e., for creating a pipeline of transformation) by specifying a comma-separated sequence of map function in the application main.

```
public abstract class AbstractMapFunction {
 public abstract GeotaggedItem apply(GeotaggedItem t);
 ...
}
```

Listing 6: Abstract class for implementing a map function using ParSoDA.

```
public class FindPoI extends AbstractMapFunction {
 public GeotaggedItem apply(GeotaggedItem g) {
  List<Poi> pois = loadRoisFromFile();
  for (Poi p : pois)
   if (GeoUtils.isContained(g.getLocation(), p.getShape())) {
    g.put(KeyNames.LOCATION_NAME, p.getName());
    return g;
   }
  return null;
 }
 ...
}
```

Listing 7: Map function *FindPoI* implemented using ParSoDA.

For the *data reduction* step, a new function is created by extending the abstract class *AbstractReduceFunction*, which has a method, namely *reduce*, that aggregates a list of items according to a given criterion. In particular, the method *reduce* receives a text key and a collection of items. The method returns a list of items as strings. For example, to implement the reduce function used in our application, i.e., *ReduceByTrajectories*, we used the *splitDataByTime* method, which included in the utility class *DataUtils* (that is part of the ParSoDA library). As shown in Listing 9, using a single line of code, the

method aggregates all the social media items posted by a single user according to a given property (i.e., *LOCATION_NAME*) and produces a list of individual sequence of locations with respect to a given time gap in days.

```
public abstract class AbstractReduceFunction {
 public abstract List<String> reduce(TextTuple key, Iterable<Text>
    items);
 ...
}
```

Listing 8: Interface to implement for creating a reduce function in ParSoDA.

```
public class ReduceByTrajectories extends extends
    AbstractReduceFunction {
  public List<String> reduce(TextTuple key, Iterable<Text> items) {
    return DataUtils.splitDataByTime(items, Metadata.LOCATION_NAME, "
    day", options.getInteger("t"));
  }
}
```

Listing 9: Reduce function ReduceByTrajectories implementation using ParSoDA.

For the *data analysis* step, a function is created by extending the *AbstractAnalysisFunction* class and implementing two abstract methods, *prepareData* and *analyzeData*, which are used to prepare data and perform analysis respectively. This class it quite generic so it can be easily extended to recall existent algorithms available in external libraries or packages, also written for other parallel runtime (es. Spark, MPI).

```
public abstract class AbstractAnalysisFunction {
 public abstract void prepareData();
 public abstract void analyzeData();
 ...
}
```

Listing 10: Abstract class for implementing a data analysis function using ParSoDA.

Finally, Listing 11 shows the abstract class that must be extended to define a custom function for the *data visualization* step. It requires to implements a method, called *visualize*, that applies a function on the data analysis results to transform them in the desired output format.

```
public abstract class AbstractVisualizationFunction {
 public abstract void visualize();
 ...
}
```

Listing 11: Abstract class for implementing a visualization function in ParSoDA.

4.3. Application code without using ParSoDA

Writing a parallel data analysis application starting from scratch usually requires a deep programming skill and writing many lines of code. In fact, designing and implementing such kind of applications poses a number of challenges to developers (e.g., communication costs, memory performance, parallelization of complex algorithms). For the study case application taken into account in this work, a developer that does not use ParSoDA has to design the entire Hadoop job, as shown in Listing 12. This requires defining and implementing several classes (e.g., input/output formats, partitioner, group and sort comparator). The whole process is not easy to do and it requires a deep knowledge of the Hadoop 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 12: The main program of the data analisys application implemented in Hadoop without using ParSoDA.

Mining data form different social media require writing hundreds lines of code to implements crawlers for collecting data from Twitter and Flickr. Moreover, since data coming from social media is heterogeneous, specific parsers are needed to convert data into a common format for the program as a whole to use. To address this issue, ParSoDA includes several parsers and define a common metadata model for representing data coming from different social media. In addition, for the most popular social media platforms, ParSoDA provides a data builder that is able to automatically infer the source of the social media items and transforms them into a common format. This means that an application written using ParSoDA is able to deal with data coming from different source without having to change the application code, which leads to an enormous saving of time for the developers.

Listing 13 shows the mapper class that a developer has to implement when ParSoDA is not used. The code is quite complex and requires many lines of code. This demonstrates that ParSoDA is able to significantly reduce the number of lines of code to be written, which allows to program complex applications more rapidly, in a more concise way, and with greater flexibility.

```
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 13: Mapper class implemented without using ParSoDA.

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 effort and skills on how the Hadoop partition phase works are required. Specifically, four classes (i.e., *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 the implementation of the reducer class, the developer must recall that the input (both key and value types) to the reducer must match the output of the mapper. The output of the reducer class must both conform to the input of the data analysis algorithm. Listing 14 shows the code of the reducer class that extracts user trajectories across locations. Compared to the code required by the ParSoDA application (i.e., one line as described in Listing 9), implementing from zero the reducer class leads to larger number of lines of code.

```
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) {
      ...
60   }
61   return ret;
62  }
63 }
```

Listing 14: Reducer class implemented without using ParSoDA.

The comparison results show that ParSoDA leads to a significant reduced number of lines of code. As reported in Table 2, it allows to achieve an overall reduction of 65% of lines of code. In particular, ParSoDA permits to save hundreds of lines of code in the data acquisition and data partition steps (where built-in functionalities are exploited), while during the data analysis and data visualization steps we have no gain since the same code (i.e., imported from external libraries) is used. The counts reported in Table 2 also takes into account additional classes that have been implemented and used in the main classes (e.g., *TextPair*).

Table 2. 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**

A key benefit for developers using ParSoDA is the possibility to set filter, map and reduction functions in the application main and then instantiate them in the relative step through a reflection mechanism. The whole process is transparent to the user that has

just to properly configure the classes to be used in the application main. This means that it is very easy for a developers to change the application behavior, e.g, by adding or removing filter/map functions without having to modify and/or recompile several classes. Moreover, following the application structure defined by ParSoDA, all the jobs created for the different steps, such as the ones in the reduction and analysis steps, are automatically chained. This means that the output of a job is automatically given in input to the next step. Instead, without using ParSoDA, a developer has to manually control the execution flow among different jobs.

We also evaluated the scalability of ParSoDA by running the data analysis application on 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 the Spark version of ParSoDA, since it resulted to be faster than the Hadoop version of the library. The analysis was carried out by analyzing a data set containing 325 GB of social media items published by Flickr and Twitter users from 2014 to 2016 referring to the center of Rome. We randomly sampled the original data set to generate a dataset containing 1024 GB. For that considered dataset, using from 25 to 300 CPU cores (i.e., from 1 to 12 worker nodes) the turnaround time has been reduced from 1.3 hour to 12 minutes. Correspondigly, the speedup achieved using ParSoDA varied from 1.9 (on 2 worker nodes) to about 7 (on 12 worker nodes).

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 HPC systems and 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 is based on implicit parallelism and it can be run on Cloud and HPC systems exploiting both Apache Hadoop and Spark. To assess the flexibility and usability of ParSoDA, we presented a social data analysis application implemented through the library for extracting sequential patterns from social media data published in Flickr and Twitter. A comparison between the application written using ParSoDA and the same one developed without using the library demonstrated that ParSoDA is able to significantly reduce the number of lines of code to be written, which allows to program complex applications more rapidly, in a more concise way, and with greater flexibility. The comparison results demonstrated a reduction of 65 % of lines of code, since the programmer only has to implement the application logic without worrying about configuring the environment and related classes.

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] Loris Belcastro, Fabrizio Marozzo, Domenico Talia, and Paolo Trunfio. Big data analysis on clouds. In Sherif Sakr and Albert Zomaya, editors, *Handbook of Big Data Technologies*, pages 101–142. Springer, December 2017. ISBN: 978-3-319-49339-8.

[2] Bo Pang and Lillian Lee. Opinion mining and sentiment analysis. *Foundations and Trends in Information Retrieval*, 2(12):1–135, 2008.

[3] Eugenio Cesario, Andrea Raffaele Iannazzo, Fabrizio Marozzo, Fabrizio Morello, Gianni Riotta, Alessandra Spada, Domenico Talia, and Paolo Trunfio. 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, 18-22 July 2016.

[4] Fabrizio Marozzo and Alessandro Bessi. Analyzing Polarization of Social Media Users and News Sites during Political Campaigns. *Social Network Analysis and Mining*, 8(1), 2018.

[5] Jeffrey Dean and Sanjay Ghemawat. Mapreduce: Simplified data processing on large clusters. In *Proceedings of the 6th Conference on Symposium on Opearting Systems Design & Implementation*, OSDI'04, pages 10–10, Berkeley, USA, 2004.

[6] Loris Belcastro, Fabrizio Marozzo, and Domenico Talia. Programming models and systems for big data analysis. *International Journal of Parallel, Emergent and Distributed Systems*, 2019.

[7] Loris Belcastro, Fabrizio Marozzo, Domenico Talia, and Paolo Trunfio. A parallel library for social media analytics. In *The 2017 International Conference on High Performance Computing & Simulation (HPCS 2017)*, Genoa, Italy, 17-21 July 2017.

[8] Loris Belcastro, Fabrizio Marozzo, Domenico Talia, and Paolo Trunfio. Appraising spark on large-scale social media analysis. In *Euro-Par Workshops*, Lecture Notes in Computer Science, pages 483–495, Santiago de Compostela, Spain, 28-29 August 2017. ISBN: 978-3-319-75178-8.

[9] Sihem Amer-Yahia, Noha Ibrahim, Christiane Kamdem Kengne, Federico Ulliana, and Marie-Christine Rousset. Socle: Towards a framework for data preparation in social applications. *Ingénierie des Systèmes d'Information*, 19(3):49–72, 2014.

[10] Álvaro Cuesta, David F. Barrero, and María D. R-Moreno. A Framework for massive Twitter data extraction and analysis. *Malaysian J. of Computer Science*, 27:1, 2014.

[11] Kristina Chodorow. *MongoDB: the definitive guide*. " O'Reilly Media, Inc.", 2013.

[12] Abid Hussain and Ravi Vatrapu. *Social Data Analytics Tool (SODATO)*, pages 368–372. Springer International Publishing, Cham, 2014.

[13] Deyu Zhou, Liangyu Chen, and Yulan He. 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.*, pages 2468–2475, 2015.

[14] Gabriella Casalino, Ciro Castiello, Nicoletta Del Buono, and Corrado Mencar. A framework for intelligent twitter data analysis with non-negative matrix factorization. *IJWIS*, 14(3):334–356, 2018.

[15] L. You, G. Motta, D. Sacco, and T. Ma. 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*, pages 96–101, Oct 2014.

[16] Konstantin Shvachko, Hairong Kuang, Sanjay Radia, and Robert Chansler. The hadoop distributed file system. In *Mass storage systems and technologies (MSST), 2010 IEEE 26th symposium on*, pages 1–10. IEEE, 2010.

[17] L. Belcastro, F. Marozzo, D. Talia, and P. Trunfio. Parsoda: high-level parallel programming for social data mining. *Social Network Analysis and Mining*, 9(1), 2019.

[18] Tom White. *Hadoop: The definitive guide*. " O'Reilly Media, Inc.", 2012.

[19] Matei Zaharia, Reynold S Xin, Patrick Wendell, Tathagata Das, Michael Armbrust, Ankur Dave, Xiangrui Meng, Josh Rosen, Shivaram Venkataraman, Michael J Franklin, et al. Apache spark: A unified engine for big data processing. *Communications of the ACM*, 59(11):56–65, 2016.

[20] Cheng Chu, Sang Kyun Kim, Yi-An Lin, YuanYuan Yu, Gary Bradski, Andrew Y Ng, and Kunle Olukotun. Map-reduce for machine learning on multicore. *Advances in neural information processing systems*, 19:281, 2007.

[21] ECMA. Ecma-262: ECMAscript Language Specification. Fifth edition. *ECMA (European Association for Standardizing Information and Communication Systems)*, 2009.

[22] Jian Pei, Jiawei Han, B. Mortazavi-Asl, Jianyong Wang, H. Pinto, Qiming Chen, U. Dayal, and Mei-Chun Hsu. Mining sequential patterns by pattern-growth: the prefixspan approach. *IEEE Transactions on Knowledge and Data Engineering*, 16(11):1424–1440, Nov 2004.