Map-Reduce for Processing GPS Data from Public Transport in Montevideo, Uruguay

Renzo Massobrio, Andrés Pías, Nicolás Vázquez, and Sergio Nesmachnow

{renzom, andres.pias, nicolas.vazquez, sergion}@fing.edu.uy
Universidad de la República, Uruguay

Abstract. This article addresses the problem of processing large volumes of historical GPS data from buses to compute quality-of-service metrics for urban transportation systems. We designed and implemented a solution to distribute the data processing on multiple processing units in a distributed computing infrastructure. For the experimental analysis we used historical data from Montevideo, Uruguay. The proposed solution scales properly when processing large volumes of input data, achieving a speedup of up to **22**× when using 24 computing resources. As case studies, we used the historical data to compute the average speed of bus lines in Montevideo and identify troublesome locations, according to the delay and deviation of the times to reach each bus stop. Similar studies can be used by control authorities and policy makers to get an insight of the transportation system and improve the quality of service.

Keywords: Map-Reduce, Big data, Intelligent Transportation Systems

1 Introduction

Intelligent Transportation Systems (ITS) are defined as those systems integrating synergistic technologies, computational intelligence, and engineering concepts to develop and improve transportation. ITS are aimed at providing innovative services for transport and traffic management, with the main goals of improving transportation safety and mobility, and also enhancing productivity [8].

ITS allow gathering large volumes of data by using sensors and devices. Smart tools that use ITS data have risen in the past years, improving the travel experiences of citizens. These tools rely on efficient and accurate data processing that poses an interesting challenge from the technological perspective. In this context, distributed computing and computational intelligence allows processing large volumes of data to be used in applications by citizens and authorities alike.

This article proposes using historical data from GPS devices installed in buses in Montevideo, Uruguay, to compute statistics to evaluate the quality of service of the transport system. The problem involves processing large volumes of data to offer real-time information to both passengers and transport authorities. For this purpose we propose a distributed computing approach using the Map-Reduce model for data processing over the Hadoop framework [6].

The article is organized as follows. Section 2 describes the problem addressed in this article and reviews the related works. Section 3 introduces the proposed model for the distributed processing and Section 4 describes the specific details of the Hadoop implementation. The experimental analysis is reported in Section 5. Finally, Section 6 presents the conclusions and main lines of future work.

2 Processing GPS data from the public transport

This section describes the problem solved in this article and reviews related works on applying Big Data and cloud computing approaches to similar problems.

2.1 Problem formulation

Given a big set of data collected from GPS devices in buses, the problem consists in computing statistical values to assess the quality of the public transportation system. The information collected by GPS includes the time and the coordinates for each bus, reported with a frequency of 10–30 seconds, which allow determining the location of each bus within its route.

The main goal of the data processing is to compute relevant metrics to assess the efficiency of the public transport system in Montevideo, for example: *i*) the real time that each bus takes to reach some important locations in the city (known as *control points* or *remarkable locations*), and *ii*) statistical information about the arriving times and delays for each of the remarkable locations (maximum, minimum, mean, mean absolute deviation, and standard deviation).

The information to report must be classified and properly organized to determine values according to different days of the week and hours in the day, which imply different passenger demands and different traffic mobility patterns.

The benefits of the proposed system are twofold: *i*) from the point of view of the users, the system provides useful information from historical data (monthly, yearly) and the current status of the public transportation in the city, to aid with mobility decisions (e.g., take a certain bus, move to a different bus stop); this information can be obtained via intelligent ubiquitous software applications and websites; *ii*) from the point of view of the city administration, the statistical information is useful for planning long-term modifications in the bus routes and frequencies, and also to address specific bottleneck situations.

A diagram of the proposed system is presented in Fig. 1. The buses upload their current location (collected by the on-board GPS unit) to a server in the cloud. The server does the Map-Reduce processing of the collected data from the different buses in real time. The results from this processing are then exposed to be consumed by the mobile app for end-users and by the monitoring application for the city government authorities.

The proposed system demands processing a high volume of data in short execution time, thus leading to a classic Big Data problem. We propose using a parallel model to do the processing, where the original data is split and distributed across different nodes to be processed independently. Finally, all the partial results from each node are combined to return the final solution.

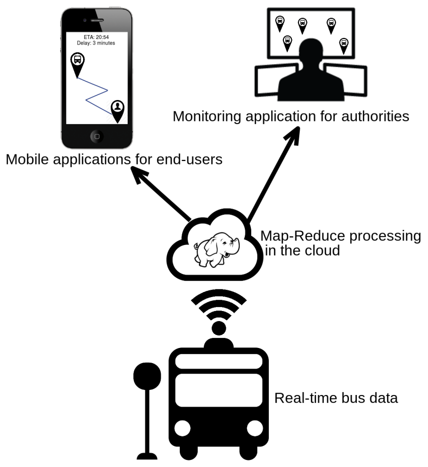
ETA: 20:54
Delay: 3 minutes
Mobile applications for end-users
Monitoring application for authorities
Map-Reduce processing in the cloud
Real-time bus data

Fig. 1. Architecture of the proposed time tracking system for buses

2.2 Related works

Several authors have applied distributed computing approaches to process large volumes of traffic data. A brief review of related works is presented next.

The advantages of using big data analysis for social transportation have been studied by Zheng et al. [10]. The authors analyze using several sources of information including vehicle mobility, pedestrian mobility, incident reports, social networking, and web logs. The advantages and limitations of using each source of data are discussed. Several other novel ideas to improve public transportation and implement the ITS paradigm are also reviewed, including *crowdsourcing* for collecting and analyzing real-time or near real-time traffic information, and *data-based agents* for driver assistance and analyzing human behavior. A conclusion on how to integrate all the previous concepts in a data-driven social transportation system that improves traffic safety and efficiency is also presented.

Other computational intelligence techniques have been recently applied to ITS design. Oh et al. [5] proposed a sequential search strategy for traffic state prediction combining a Vehicle Detection System and k nearest neighbors (kNN), which outperforms a traditional kNN approach, computing significantly more accurate results while maintaining good efficiency and stability properties. Shi and Abdel-Aty [7] applied the random forest data mining technique and Bayesian inference to process large volumes of data from a Microwave Vehicle Detection System to identify the contributing factors to crashes in real-time, concluding that congestion has the most impact on rear-end crashes. Ahn et al. [1] applied Support Vector Regression (SVR) and a Bayesian classifier for building a real-time traffic flow prediction system. The performance of the proposed method is studied on traffic data from South Korea, showing that the SVR-based estimation outperformed a traditional linear regression method in terms of accuracy.

Chen et al. [2] applied kNN and a Gaussian regression on Hadoop to efficiently predict traffic speed using historical ITS data, weather conditions, and other events. The evaluation over a real scenario based on the I5N road section (US) considered speed, flow, occupancy, and visibility data (from weather stations nearby). The proposed method predicted traffic speed with an average forecasting error smaller that 2 miles per hour and the execution time was reduced in 69% in a cluster infrastructure against running in a single machine. Xia et al. [9] studied the real-time short-term traffic flow forecasting problem, applying kNN in a distributed environment using Hadoop. The evaluation considered data from over 12000 GPS-equipped taxis in the city of Beijing, China. The proposed algorithm allows reducing the mean absolute percentage error between 8.5% to 11.5% on average over three existing kNN-based techniques. Additionally a computational efficiency of 0.84 was achieved in the best case.

The related works propose several strategies for using big data analysis and computational intelligence to improve ITS. However, few works focus in the interests of users. The research reported in this article contributes with a specific proposal to monitor and improve the public bus transportation in Montevideo, Uruguay, considering the point of view of both users and administrators.

3 The proposed solution

This section describes the proposed solution for processing historical GPS data from buses in the public transport system.

3.1 Design and architecture

The problem is decomposed in two sub-problems: *i*) a pre-processing to properly prepare the input data for the next phase, and *ii*) the parallel/distributed approach to compute the statistics of the public transportation system. A master-slave model is used to define and organize the control hierarchy and processing. Fig. 2 presents a conceptual diagram of the proposed system.

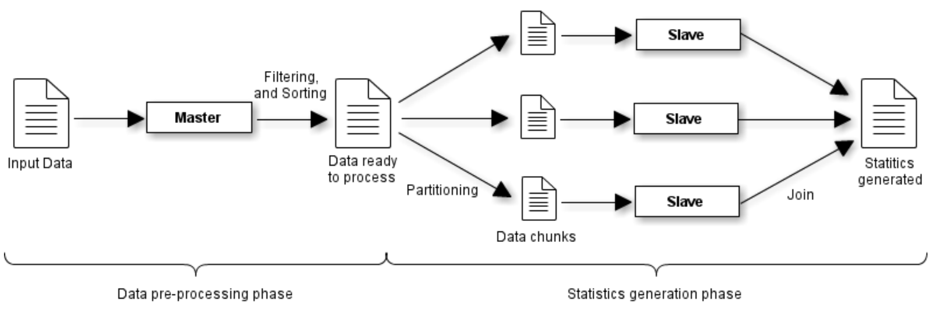

Fig. 2. Conceptual diagram of the proposed application

In the pre-processing phase, the master process filters the data to select the information useful to compute the statistics. The data processing phase applies a data-parallel domain decomposition strategy for parallelization. The available data from the previous phase is split in chunks to be handled by several processing elements. The master process is in charge of controlling the system, performing the data partition, and sending the chunks to slaves for processing. Each slave process receives a subset of the data from the master. The group of slaves processes collaborate in the data processing, generating the expected statistical results, following a single program multiple data (SPMD) model.

3.2 Strategy for data processing: algorithmic description

The input data correspond to the GPS coordinates sent by each bus in operation, during each travel. Lines in the input file represent positions recorded by a bus during a given route. The *line code* is a unique identifier for each bus line. In turn, to distinguish different travels for the same bus line, the file has a self-generated numeric field, which identifies a particular *travel number* for each line.

Pre-processing stage. The pre-processing prepares the data, discriminating those records that do not contain useful information for computing the statistics, and classifying/ordering the useful records. Three phases are identified:

1. *Filtering*: filters the data according to the statistics to compute. Two relevant cases are considered: *i*) *discarding non-useful data*, which is present in the raw data files and *ii*) *filtering ranges*, according to the time range received as parameter to determine the data selection for computing the statistics.
2. *Time zone characterization*: identifies the time zone for each record containing useful information. Due to the variations on the traffic patterns, the time zone must be taken into account to compute and analyze the generated statistics. is relevant to compute and analyze statistics generated due to variations of this factor determines very different values to be processed. We consider three time frames in the study: *morning* (between 04:00 and 12:00), *afternoon* (between 12:01 and 20:00), and *night* (between 20:01 and 4:00).
3. *Sorting*: sorts the records according to the bus service identification (*line number*), the route identification (route *variant*), and the timestamp of the record. These three fields define a processing key, which is needed to compute the time differences between the departing time for each bus and the time to reach each one of the control points (see Section 4).

After applying the pre-processing stage, the master process has the filtered data, to be used as the input data for the processing phase. The data consist in a set of records containing six fields: *bus line number*, *travel variant number*, *travel number*, *timestamp*, *timeframe* and *bus stop*.

Statistics generation stage. This stage is organized in four phases:

1. *Data partitioning and distribution*: The master process divides the dataset of useful GPS records and distributes the resulting subsets among the slave processes. Each subset includes a group of records associated with the same *bus line number* and *travel variant number*, sorted according to the criteria applied in the pre-processing stage. Statistics associated with the same *bus line number* and *travel variant number* are processed in the same slave.
2. *Computing temporal distances.* Each slave processes the subset sent by the master process, by splitting each record into different fields, to create new data. For each pair (*bus line number*, *travel variant number*), the distances between different points on the bus route are calculated iterating through a date-ordered list containing the distance values. A temporal counter is initialized with the value 0 every time that a new occurrence of a travel number is found in the subset containing the GPS data to be handled by each slave process. That time is considered as the initial travel time, and the slave uses it to track each remarkable location or bus stop, computing the time between the timestamp and the initial time (i.e., the *relative time*). The computed relative times are then filtered by *timeframe* and by *remarkable location* (besides the original filter by *bus line number* and *travel variant number*). The generated results are stored in memory, to be available for using them in the next phase, grouped by the four fields mentioned before.
3. *Statistics generation.* Data are reduced into results and statistics are computed. For each occurrence, an iteration over the calculated distances is performed to compute several metrics, including the maximum and minimum differences between times, the average times, and the standard deviation on time values, computed considering the relative time to reach control points and remarkable locations for each bus line, and filtering by the different time frames considered. The output values are grouped and ordered by *bus line number*, *travel variant number*, *remarkable location* and *timeframe.*
4. *Return results to the master.* Finally, the slaves return the partial results to the master, who groups them and prompts the final results to the user.

4 Implementation of the Map-Reduce application

The proposed parallel/distributed system for traffic data processing is implemented using a Map-Reduce approach in Hadoop. The application fits in the Map-Reduce model, because no communications are required between slave processes, and the only communications between master and slaves are performed for the initial phase of data distribution and the final phase to report the results.

4.1 The Map-Reduce implementation

The application applies the standard Map-Reduce engine in Hadoop, using one master node and several slave nodes. The master node uses the *JobTracker* process to send jobs to different *TaskTracker* processes associated to the slave nodes.

When the slaves finish processing, each *TaskTracker* sends the results back to the *JobTracker* in the master node. An overview of the application and the data flow is presented in Fig. 3. The main implementation details are described next.

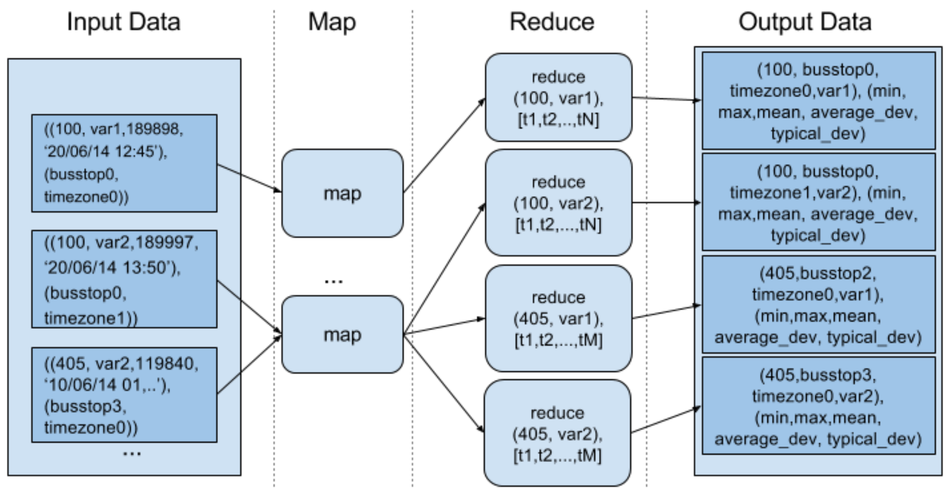
Input Data
Map
Reduce
Output Data
((100, var1,189898, '20/06/14 12:45'), (busstop0, timezone0))
((100, var2,189997, '20/06/14 13:50'), (busstop0, timezone1))
((405, var2,119840, '10/06/14 01,..'), (busstop3, timezone0))
...
map
...
map
reduce (100, var1), [t1,t2,..,tN]
reduce (100, var2), [t1,t2,...,tN]
reduce (405, var1), [t1,t2,...,tM]
reduce (405, var2), [t1,t2,...,tM]
(100, busstop0, timezone0,var1), (min, max,mean, average_dev, typical_dev)
(100, busstop0, timezone1,var2), (min, max,mean, average_dev, typical_dev)
(405,busstop2, timezone0,var1), (min,max,mean, average_dev, typical_dev)
(405,busstop3, timezone0,var2), (min,max,mean, average_dev, typical_dev)

Fig. 3. Diagram of the proposed Map-Reduce implementation

Pre-processing stage. The pre-processing stage involves the traditional phases usually applied when using the Hadoop framework: *Splitting*, *Mapping*, and *Shuffling and Sorting*. All these tasks are performed by the Hadoop master process.

Splitting. The splitting phase corresponds to the distribution of records to the master processes, represented by the arrows from Input Data to Map in Fig. 3. Two instances of the `FileInputFormat` and `RecordReader` classes in Hadoop were implemented to filter useful data and generate the input (`InputSplit`) for the mapper processes. The selected records are converted to appropriate datatypes to be used in the statistcs generation stage (e.g., numerical data are converted to `long` or `int`, dates are converted to `timestamp`, etc.).

Mapping. The mapping phase is clearly identified in the diagram in Fig. 3. Each mapper receive a subset of data (data block) to process. Let X be the size of the data to process and b the size of the data blocks, the number of mappers on execution is defined by X/b. A data filtering is applied by each mapper.

The `FilteringInputFormat` class (Listing 1) overrides `createRecordReader` to return object types `RecordReader<KeyBusCompound,BusInfo>`. Both `KeyBusCompound` and `BusInfo` extends `WriteComparable`, and they are used as the <key,value> pair for records sent to mappers. `KeyBusCompound` is a compound key that includes all the fields needed to identify a bus trip (public name, travel variant, travel number, and timestamp). `BusInfo` includes values for remarkable locations, timeframe, and travel number and timestamp, needed to apply a secondary sorting (see *Sorting*).

```
public class FilteringInputFormat
extends FileInputFormat<KeyBusCompund,BusInfo> {
    @Override
    public RecordReader<KeyBusCompound,BusInfo>
    createRecordReader(InputSplit inputsplit,
    TaskAttemptContext taskattemptcontext)
    throws IOException, InterruptedException{
        return new FilteringRecordReader();
    }
}
```

Listing 1. FilteringInputFormat classs

Being $\{L\}$ the set of input lines received in the splitting phase, the Hadoop framework creates a number of `FilteringInputFormat` instances according to the number of input splits on the input file. Each `FilteringInputFormat` instance p operates on a data subset $\{L\}^p$.

The code of class `FilteringRecordReader` code is large; Listing 2 presents the `nextKeyValue` method as an example of the filtering logic used to return the next register to be processed by mappers. Let {T} be the set of resulting filtered data. In `nextKeyValue`, only those lines whose line numbers are between *start* and *end* values are processed: for each record on the input, an integrity check is performed (to discard records without the expected format and those not included in the range) and if not discarded, a <KeyBusCompound, BusInfo> record is added to the resulting set. The output is a list containing $\{T\}^p$ records, with the format <(public_name, travel_variant, travel_number, timestamp), info>.

```
repeat=true
pos=getActualPosition()
do while (repeat and pos<end)
    line = ReadLine();
    fields = line.split(',')
    // Map fields to local variables
    if (fields.length!=9)
       repeat=false
    elseif ( not validateFormat(fields) )
       repeat=false
    elseif ( date_i<initial_date or date_i>end_date )
       repeat=false
    else
       key_i = KeyBusCompound(bus_line_number_i,travel_variant_i,
           travel_number_i,date_i)
       value_i = BusInfo(travel_number_i,date_i,point_i,timeframe_i)
       repeat=true
end do

if ((pos>=end) and (repeat)
    return false
else
    return true
```

Listing 2. Filtering logic applied in the pre-processing stage

Shuffling and Sorting. On a typical Map-Reduce solution, Sorting, Shuffling and Partioning come after the Map stage. We decided to apply a Secondary Sorting [6] in order to deliver ordered values to each reducer to calculate temporary distances. The common practice in Map-Reduce applications is sorting keys, but in the case of our system the Secondary Sorting is needed because we need to sort keys and values (i.e., the known *value-to-key conversion* procedure). Specific instances of `Partitions` and `WritableComparator` were defined:

- `NaturalKeyPartitioner`: extends the `Partitioner` class, to define the specific partitioning method.
- `NaturalKeyGroupingComparator`: groups records associated to the same key (<public_name, travel_variant>), to be sent to the same reducer.
- `CompKeyComparator`: used to order key values sent to reducers, according to the compound key <public_name, travel_variant, travel_number, date>

Statistic generation stage. The statistic generation stage is performed by Hadoop reduce processes, which correspond to slave processes in the conceptual algorithmic description (each one with the three phases identified in Section 3).

Data partitioning and distribution. This phase corresponds to the data sent from Map to Reduce in the diagram in Fig. 3. By default, the Hadoop framework distributes keys to different reducers applying a hashmap partitioning. This distribution mechanism does not guarantee an appropriate load balancing, because reducers do not receive equally-size subsets to process. Furthermore, the results produced by reducers will not be ordered, as it is desirable for the reports to be delivered to the users in the proposed application. For these reasons, we implemented a specific partitioning method on the `NaturalKeyPartioner` class. Each instance of `NaturalKeyPartioner` uses a TreeMap hash to decide the reducer to send those records associated to a specific bus line number and travel variant. The TreeMap structure is dynamically generated in the main program, taking into account the number of reducers and a CSV file containing ordered unique keys (line,variant).

Reduce. The previous phase allows guaranteeing that each reducer process receives all the data related to a given bus line and a travel variant, and that all values associated to keys are ordered. Each Reducer receives tuples with the format: <(public_name$_i$, travel_variant$_i$), [info$_{i1}$, info$_{i2}$, info$_{i3}$, ...]> The input key type is `KeyBusCompound` and the type of the input values is `BusInfo`. A reduce function is executed on each key (public_name, travel_variant) on set {T}. In a first step, the initial times are determined for each particular bus travel; from this result, the relative times between remarkable locations in the travel (see a description in Listing 3). Each reduce function iterates through the list of input values [info$_{i1}$, info$_{i2}$, ..., info$_{iN}$]. Data is stored temporarily in a TreeMap structure, in which `KeyStatistics` is used as key and `LongWritable` as values. `KeyStatistics` is defined as (bus_line$_i$, control_point$_i$, timeframe$_i$, variant$_i$) and `LongWritable` values represent the time differences.

```
actual_travel=0
for each row i on input
  if actual_travel != travel_number$_i$
    actual_travel = travel_number$_i$;
      start = date$_i$
  else
      time_diference = date$_i$ - start
      timeframe <- calculated using date$_i$
        value$_i$ <- Long(time_diference)
        statisticsKey$_i$ <- KeyStatistics(bus_line$_i$,
        travel_variant$_i$, timeframe$_i$, control_point$_i$)
        Add value$_i$ to key$_i$ values in TreeMap structure
end for
```

Listing 3. Reduce function

After that, a second function on the reducer receives each <KeyStatistic, Long> pair and computes the statistic values from the previous partial results. In the cases of study reported in this article, we compute the maximum, minimum, arithmetic mean, mean absolute deviation, and standard deviation of times for each bus line, control point, timeframe, and variant. The reducers output is <(public_name$_i$, variant$_i$, timeframe$_i$, control_point$_i$), (min_time$_i$, max_time$_i$, mean$_i$, mean_deviation$_i$, mean_deviation$_i$)>. These pairs are represented as text, key, and value, to prompt results to user.

4.2 Fault tolerance

The automatic fault tolerance mechanism included in Hadoop is used. Additionally, some features are activated to improve fault tolerance for the proposed application: *i*) the feature that allows discarding corrupt input lines is enabled, to be used in those cases where a line cannot be read (the impact of discarding corrupt input lines is not significant, because the system is oriented to compute statistics and estimated values); and *ii*) the native replication mechanism in HDFS was activated, to keep data replicated in different processing nodes.

5 Experimental evaluation

This section describes the experimental evaluation of the proposed system. The computational platform and the problem instances generated from the historical data are described. After that, the computational efficiency results are reported. Finally, case studies are presented for a specific bus line.

5.1 Experiments setup

Computational platform. The experimental evaluation was performed over the cloud infrastructure provided by Cluster FING, Universidad de la República, Uruguay [4], using AMD Opteron 6172 Magny Cours (24 cores) processors at 2.26 GHz, 24 GB RAM, and CentOS Linux 5.2 operating system.

Problem instances and data. Several datasets are used to define different test scenarios conceived to test the behavior of the system under diverse situations, including different input file sizes, different time intervals, and using different number of map and reduce processes. We work with datasets containing 10 GB, 20 GB, 30 GB, and 60 GB, and also different time intervals (3 days, and 1, 2, 3, and 6 months), with real GPS data from buses in Montevideo, provided by the local administration *Intendencia de Montevideo*. The input data file to use in each test is stored in HDFS. To better exploit the parallel processing, more mappers than HDFS blocks must be used when splitting the file. Considering an input file of size X MB and HDFS blocks of size Y MB, the algorithm needs using at least X/Y mappers. Hadoop uses the input file size and the number of mappers created to determinate the number of splits on the input file.

Metrics. We apply the traditional metrics to evaluate the performance of parallel algorithms: the *speedup* and the *efficiency*. The speedup evaluates how much faster is a parallel algorithm than its sequential version. It is defined as the ratio of the execution times of the sequential algorithm (T_1) and the parallel version executed on N computing elements (T_N) (Eq. 1). The ideal case for a parallel/distributed algorithm is to achieve linear speedup ($S_N = N$). However, the common situation is to achieve sublinear speedup ($S_N < N$), due to the times required to communicate and synchronize the parallel/distributed processes or threads. The efficiency is the normalized value of the speedup, regarding the number of computing elements used for execution (Eq. 2). This metric allows comparing algorithms executed in non-identical computing platforms. The linear speedup corresponds to $E_N = 1$, and in usual situations $E_N < 1$.

$$S_N = \frac{T_1}{T_N} \qquad (1) \qquad\qquad E_N = \frac{S_N}{N} \qquad (2)$$

5.2 Experimental results

We evaluated the computational efficiency of the proposed distributed solution and also the correctness to produce useful information for users and administrators. The main results are reported below.

Computational efficiency analysis. Table 1 reports the computational efficiency results for the proposed application when varying the size of the input data (#I), days (#D), number of mapper (#M) and reducer (#R) processes. Mean values computed over five independent executions are reported for each metric. Times are in seconds.

The results in Table 1 demonstrate that the distributed algorithm allows significantly improving the efficiency of the sequential version, especially when processing large volumes of data. The better speedup values are obtained when processing the 60GB input file, i.e., **22.16**, corresponding to a computational efficiency of **0.92**. The distributed implementation allows reducing the execution time from about 6 hours to 14 minutes when processing the 60GB input data file. This efficiency result is crucial to provide a fast response to specific situations and to analyze different metrics and scenarios for both users and administrators.

#I	#D	#M	#R	T_1(s)	T_N(s)	S_N	E_N
10	3	14	8	1333.9	253.1	5.27	0.22
10	3	22	22	1333.9	143.0	9.33	0.39
10	30	14	8	2108.6	178.0	11.84	0.49
10	30	22	22	2108.6	187.3	11.26	0.47
20	3	14	8	2449.0	351.1	6.98	0.29
20	3	22	22	2449.0	189.8	12.90	0.54
20	30	14	8	3324.5	275.6	12.06	0.50
20	30	22	22	3324.5	238.8	13.92	0.58
20	60	14	8	4762.0	300.8	15.83	0.66
20	60	22	22	4762.0	264.7	17.99	0.75
30	3	14	8	3588.5	546.9	6.56	0.27
30	3	22	22	3588.5	179.6	19.99	0.83
30	30	14	8	5052.9	359.6	14.05	0.59
30	30	22	22	5052.9	281.1	17.98	0.75
30	60	14	8	5927.9	383.4	15.46	0.64
30	60	22	22	5927.9	311.4	19.04	0.79
30	90	14	8	7536.9	416.6	18.09	0.75
30	90	22	22	7536.9	349.2	21.58	0.90
60	3	14	8	7249.6	944.0	7.68	0.32
60	3	22	22	7249.6	362.1	20.02	0.83
60	60	14	8	10037.1	672.6	14.92	0.62
60	60	22	22	10037.1	531.4	18.89	0.79
60	90	14	8	11941.6	709.6	16.83	0.70
60	90	22	22	11941.6	648.9	18.40	0.77
60	180	14	8	19060.8	913.7	20.86	0.87
60	180	22	22	19060.8	860.3	22.16	0.92

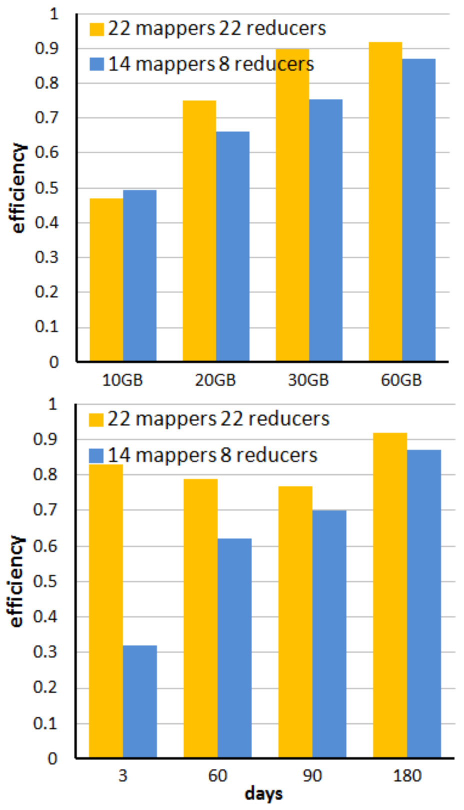

Table 1. Results of the experimental analysis: computational efficiency

Using 22 mappers and 22 reducers allows obtaining the best efficiency, improving in up to **15**% the execution time (**9**% in average) over the ones demanded when using 14 mappers and 8 reducers. Working on small problems size causes that data to be partitioned is small pieces, generating low loaded processes and not improving notably over the execution time of the sequential algorithm.

The efficiency analysis also determines that the Map and Reduce phases have similar execution times and they reach the max CPU usage (above 97% at every moment). These results show that the load balance efforts in the proposed algorithm prevents a majority of idle or low-loaded mappers and reducers.

Case of study: average speed and troublesome locations. We report the analysis of a relevant study for the public transport in Montevideo: the calculation of the average speed of buses and troublesome locations in the city.

Fig. 4 shows a report extracted from the analysis of speed of buses to identify troublesome locations. Results correspond to bus line 195 in the night. Speed and delays values are computed according to six months of historical GPS records and the times to reach each bus stop, compared against the theoretical scheduled times, as reported in the website of the Transport System of Montevideo (Sistema de Transporte Metropolitano, STM) [3].

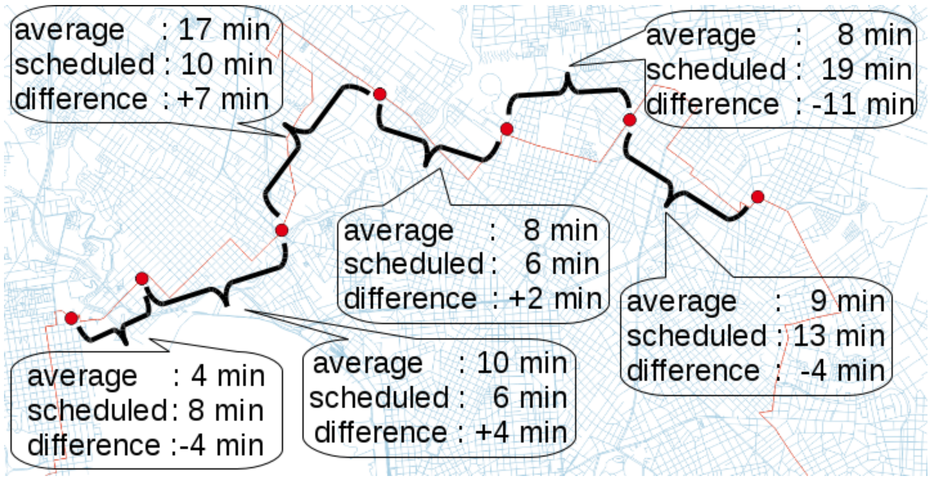

Fig. 4. Average delay for bus line 195 in the night, using 6 months of historical data

These results can be obtained in real time using the distributed algorithm, allowing a fast response to specific problems. In addition, the information can be reported to users via mobile ubiquitous applications.

6 Conclusions and future work

This article describes a Big Data analysis using distributed computational intelligence to process historical GPS data to compute quality-of-service metrics for the public transportation system in Montevideo, Uruguay.

An intelligent system for data processing was conceived, applying the Map-Reduce paradigm implemented over the Hadoop framework. Specific features were included to deal with the processed data: the proposed implementation allows filtering and selecting useful information to compute a set of relevant statistics to assess the quality of the public transportation system. An application-oriented load balancing schema was also implemented.

The experimental analysis focused on evaluating the computational efficiency and the correctness of the implemented system, working over several scenarios built by using real data form GPS data collected in 2015 in Montevideo. The main results indicated that the proposed solution scales properly when processing large volumes of input data, achieving a speedup of **22.16** when using 24 computing resources, when processing the largest input files.

As a case study, we computed the average speed of bus lines in Montevideo using the available historical data, to identify troublesome locations in the public bus network, according to the delay and deviation of the times to reach each bus stop. This kind of studies can be used by control authorities and policy makers to better understand the transportation system infrastructure and to improve the quality of service. The information can also be incorporated to mobile applications for passengers to improve the travel experience.

In this article we processed the GPS bus data gathered during 6 months of 2015, however the proposed distributed architecture would scale up efficiently when processing larger volumes of data, as shown in the experimental analysis. The city government collects the GPS data periodically, so it is possible to incorporate additional data in order to get even more accurate statistics. Furthermore, the Uruguayan government handles several other ITS and non-ITS data sources (including GPS data for taxis, mobile phone data, ticket sale data, special events in the city) which could be easily incorporated to the proposed model to get a holistic understanding of mobility in the city.

The main lines for future work are oriented to further extend the proposed system, including the calculation of several other important indicators and statistics to assess the quality of the public transportation. Relevant issues to include are the construction of origin-destination matrices for public transport, the evaluation of bus frequencies (and dynamic adjustment), etc. The proposed approach can also be extended to provide efficient solutions to other smart city problems (e.g., pedestrian and vehicle fleets mobility, energy consumption, and others). Using other distributed computation frameworks (such as Apache Storm) is also a promising idea to better exploit the real-time features of the proposed system.

References

1. J. Ahn, E. Ko, and E. Yi Kim. Highway traffic flow prediction using support vector regression and bayesian classifier. In *International Conference on Big Data and Smart Computing*, pages 239–244, 2016.
2. X. Chen, H. Pao, and Y. Lee. Efficient traffic speed forecasting based on massive heterogenous historical data. In *IEEE International Conference on Big Data*, pages 10–17, 2014.
3. Intendencia Municipal de Montevideo. *Plan de Movilidad*. Montevideo, 2010.
4. S. Nesmachnow. Computación científica de alto desempeño en la Facultad de Ingeniería, Universidad de la República. Revista de la Asociación de Ingenieros del Uruguay, 61:12–15, 2010.
5. S. Oh, Y. Byon, and H. Yeo. Improvement of search strategy with k-nearest neighbors approach for traffic state prediction. *IEEE Trans. Intell. Transport. Syst.*, 17(4):1146–1156, 2016.
6. M. Parsian. *Data Algorithms, Recipes for Scaling Up with Hadoop and Spark*. O'Reilly Media, 2015.
7. Q. Shi and M. Abdel-Aty. Big data applications in real-time traffic operation and safety monitoring and improvement on urban expressways. *Transportation Research Part C: Emerging Technologies*, 58:380–394, 2015.
8. J. Sussman. *Perspectives on Intelligent Transportation Systems (ITS)*. Springer Science + Business Media, 2005.
9. D. Xia, B. Wang, H. Li, Y. Li, and Z. Zhang. A distributed spatialtemporal weighted model on mapreduce for short-term traffic flow forecasting. *Neurocomputing*, 179:246–263, 2016.
10. X. Zheng, W. Chen, P. Wang, D. Shen, S. Chen, X. Wang, Q. Zhang, and L. Yang. Big data for social transportation. *IEEE Trans. Intell. Transport. Syst.*, 17(3):620–630, 2016.