J Supercomput (2017) 73:514–529
DOI 10.1007/s11227-016-1896-3

Cloud implementation of the K-means algorithm for hyperspectral image analysis

Juan Mario Haut[1] · **Mercedes Paoletti**[1] · **Javier Plaza**[1] · **Antonio Plaza**[1]

Published online: 18 October 2016

Abstract Remotely sensed hyperspectral imaging offers the possibility to collect hundreds of images, at different wavelength channels, for the same area on the surface of the Earth. Hyperspectral images are characterized by their large volume and dimensionality, which makes their processing and storage difficult. As a result, several techniques have been developed in previous years to perform hyperspectral image analysis on high-performance computing architectures. However, the application of cloud computing techniques has not been as widespread. There are many potential advantages in exploiting cloud computing architectures for distributed hyperspectral image analysis. In this paper, we present a cloud implementation (developed using Apache Spark) of the popular K-means algorithm for unsupervised hyperspectral image clustering. The experimental results suggest that cloud architectures allow for the efficient distributed processing of large hyperspectral image data sets.

Keywords Hyperspectral imaging · Cloud computing · K-means clustering

✉ Javier Plaza
jplaza@unex.es

Juan Mario Haut
juanmariohaut@unex.es

Mercedes Paoletti
mpaolett@alumnos.unex.es

Antonio Plaza
aplaza@unex.es

[1] Department of Technology of Computers and Communications, University of Extremadura, Escuela Politecnica, Avda. de la Universidad s/n, Cáceres, Spain

1 Introduction

Hyperspectral images comprise hundred of spectral bands collected at nearly contiguous wavelengths, thus imposing significant requirements in terms of storage and data processing [12]. These requirements have increased exponentially with the technological advances in satellite and airbone remote sensing [6], leading to the development of high-dimensional hyperspectral data repositories [13].

The increased availability of new hyperspectral missions is now generating an almost continuous stream of multi/hyperspectral data [9], and this has introduced important challenges for scalable and efficient hyperspectral data processing in different application domains [11]. For instance, the NASA Jet Propulsion Laboratory's Airbone Visible/Infrared Imaging Spectrometer (AVIRIS) [4] has a data collection rate of 2.5 MB/s (nearly 9 GB/h). The space-borne Hyperion instrument [11] collects data at a rate of almost 71.9 GB/h (over 1.6 TB/day). New satellite missions will be soon in operation, such as the Environmental Mapping and Analysis Program (EnMAP),[1] exhibiting similar data collection rates. Hyperspectral data repositories are becoming increasingly massive and often distributed among several geographic locations, which makes it difficult to meet the storage and computational requirements of large-scale hyperspectral data processing applications without resorting to distributed computing facilities.

In recent years, cloud computing platforms have been increasingly adopted for remotely sensed data processing [15]. The cloud is now a standard for distributed computing due to its advanced capabilities for internet-scale computing, service-oriented computing, and high-performance computing. The use of cloud computing for the analysis of large hyperspectral data repositories can be considered a natural solution and an evolution of previously developed techniques for other kinds of computing platforms [8]. However, there are few efforts in the recent literature oriented to the exploitation of cloud computing infrastructure for hyperspectral imaging techniques in general, and for unsupervised clustering algorithms in particular.

Clustering can be defined as a segmentation process in which pixels are assigned into a group that represents a specific land-cover class [5]. The main advantage of clustering is that there is no need for labeled samples which are difficult and expensive to obtain in remote sensing scenarios [10]. In this regard, clustering offers an alternative to supervised classifiers that has been widely used in various fields. However, clustering is also a very challenging task due to the large spectral variability and complex spatial structures present in hyperspectral images. A widely used family of clustering algorithms is represented by centroid-based clustering methods such as K-means [5], which assumes that similar pixels form clusters in feature space. When applied to hyperspectral images, these methods can provide satisfactory results but are hampered by their large computational complexity.

In this paper, we explore for the first time in the literature the possibility of using a distributed framework for clustering of massive hyperspectral images based on cloud computing architectures. We use unsupervised clustering as a case study, focusing

[1] http://www.enmap.org/.

on the popular K-means algorithm to demonstrate the applicability of utilizing cloud computing technologies to efficiently perform distributed parallel processing of hyperspectral data and accelerate computations.

The remainder of the paper is organized as follows. Section 2 presents the distributed framework design that will be used in our implementation. Section 3 presents the original K-means algorithm. Section 4 describes distributed and multicore implementations of the K-means algorithm. Section 5 presents the experimental validation of the considered implementations. Section 6 concludes the paper with some remarks and hints at future research lines.

2 Distributed framework design

To develop a distributed framework for implementing the K-means algorithm on cloud computing architectures, two main issues need to be addressed: (1) the distributed programming model, and (2) the computing engine.

For distributed programming, we resort to the MapReduce model [15], taking full advantage of the high-performance capabilities provided by cloud computing architectures. In this model, a task is processed by two distributed operations: map and reduce. The datasets are organized as key/value pairs, and the map function processes a key/value pair to generate a set of intermediate pairs, dividing a task into several independent subtasks to be run in parallel. The reduce function is in charge of processing all intermediate values associated with the same intermediate key, then collecting all the subtask results to gather the result for the whole task.

Regarding the distributed computing engine, a first solution considered was Apache Hadoop[2] due to its reliability and scalability, as well as its completely open source nature. However, Apache Hadoop only supports simple one-pass computations and is generally not appropriate for iterative algorithms such as K-means. Apache Spark[3] is a new computing engine for large-scale data processing on cloud computing architectures, which implements a fault-tolerant abstraction for in-memory cluster computing, and provides fast and general data processing on large distributed platforms. It not only supports simple one-pass computations, but can also be extended to the case of multi-pass, iterative algorithms.

With the aforementioned issues in mind, the design of our distributed parallel framework for hyperspectral data clustering using Apache Spark is graphically sketched in Fig. 1. As shown by Fig. 1, the considered cloud architecture has two main parts:

– *The hardware zone* It contains the physical machines that support our virtual machines, which are created in the OpenStack[4] platform, a cloud operating system that controls large pools of compute, storage, and networking resources throughout a datacenter, all managed through a dashboard that gives administrators control while empowering their users to provision resources through a web interface.

[2] http://hadoop.apache.org.

[3] http://spark.apache.org/.

[4] https://wiki.openstack.org/wiki/Main_Page.

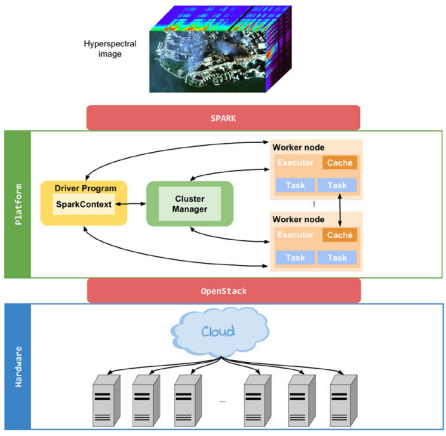


Fig. 1 Integrated Apache Spark and OpenStack framework used for the implementation of K-means

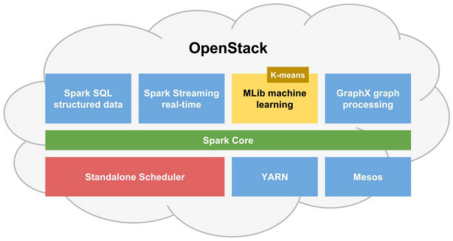


Fig. 2 Description of the Apache Spark architecture used in our experiments

– *The platform zone* The Apache Spark framework in Fig. 1 is installed over a set of Ubuntu Linux virtual machines, created by OpenStack. Our cluster has various types of nodes. The K-means algorithm is designed using the MLib machine learning library,[5] and the implementation is embedded into a joint Apache Spark and OpenStack framework (see Fig. 2). When we launch an instance of K-means, the master node manages the resources of the cluster and the slaves (workers) perform individual tasks on the data. The driver node divides the work into tasks, and the master node coordinates the allocation of tasks so that all the task are executed by the worker nodes, following the MapReduce model.

3 The K-means clustering algorithm

K-means is one of the most widely used unsupervised algorithms to group data in a specified number of clusters. The procedure begins with a set of data or observations, $X = [x_1, x_2, \ldots, x_n]$, in $\mathbb{R}^d$ (so $x_i = [x_{i1}, x_{i2}, \ldots, x_{id}]$, with $i = 1, 2, \ldots, n$), that needs to be grouped into a number of *clusters* ($k <= n$). Iteratively, K-means calculates the centers of the k groups, optimizing the error of each group as follows:

[5] https://spark.apache.org/docs/latest/mllib-guide.html.

$$\min \sum_{j=1}^{k} \sum_{i=1}^{n_k} \| x_i^j - C_j, \|^2 \tag{1}$$

where $\| x_i^j - C_j \|^2$ is the distance between a data point x_i^j of the cluster j (n_k is the number of observations within each cluster) and the cluster center C_j.

Besides its high computational cost (execution time can be exponential in the worst case), the K-means algorithm strongly depends on the choice of the initial centers (it can converge to a local minimum). So, a proper initialization will result in a final best solution. To obtain a set of good initial cluster centers, several methods have been proposed. One of them is the K-means++ method [1]. This algorithm obtains a set of k initial centers which are generally very close to the final solution by the following five steps:

1. An initial point is chosen from the set of samples $X = [x_1, x_2, \ldots, x_n]$ by an uniform random variable. This point c_1, is the first center.
2. For each sample x_i, the distance $D(x)$ between x_i and the center c_1 is calculated.
3. Then, a new candidate to become center is randomly selected using the probability-weighted distribution $\frac{D(x)^2}{\sum_{i=0}^{k} D(x_i)^2}$.
4. Steps 2 and 3 are repeated until k initial centers have been selected.
5. Once the k initial centers have been chosen, we apply the standard K-means algorithm.

Despite the fact that it is a standard algorithm which provides a good solution, K-means++ has two main problems: (1) it is not scalable, since its initialization needs k passes over the whole dataset (which is not recommended for very large images), and (2) the choice of the next set of centers depends on the current set of centers.

4 Multicore/distributed implementation of K-means

In this section, we describe two different implementations of the original K-means++ algorithm: a distributed implementation and a multicore implementation.

4.1 Distributed implementation on Apache Spark

Apache Spark implements a parallel version of the K-means++ method, called K-means|| [2] that takes advantage of the MapReduce model of computation. K-means|| consists of a modification of the original K-means++ that manages noisy outliers and also reduces the number of iterations. In fact, its operation is very similar to the original K-means++. The difference is in the initialization method of the set of centroids. Like K-means++, K-means|| starts by randomly choosing a point with uniform distribution as the first centroid, $C \leftarrow c_1$, and computes the initial cost of the clustering after this selection: $\psi = \phi x(C)$, where $\phi x(C) = \sum_{x \in X} d^2(x, C)$. It then samples each $x \in X$ with probability $p_x = \frac{l \cdot d^2(x,C)}{\phi x(C)}$ in $\log \psi$ iterations, adding the sampled points to C. The expected number of points chosen in each iteration is l, that is the oversampling

factor. At the end of the algorithm, $l \log \psi$ points are obtained, which are clustered in k centers. This process is illustrated in Algorithm 1.

Algorithm 1 K-means|| algorithm

1: **procedure** K- MEANS||(k, l) ▷ $k \rightarrow$ number of clusters, $l \rightarrow \Theta(K)$
2: $C \leftarrow$ uniform_rand(X)
3: $\psi \leftarrow \phi_X(C)$
4: **for** $O(\log \psi)$ **do**
5: $C' \leftarrow x \in X$ with $p(x) = \frac{l \cdot D(x)^2}{\phi_X(C)}$ ▷ Probability-weighted distribution
6: $C \leftarrow C \bigcup C'$
7: **end for**
8: For $x \in C$, set w_x to be the number of points in X closer to x than any other point in C
9: Recluster (C, k)
10: **end procedure**

To use the K-means|| method in Apache Spark, the user must specify the following parameters: number of desired clusters, k; maximum number of iterations that the algorithm can be executed, $maxIterations$; and number of times to execute the algorithm completely, $runs$.[6] The parameter $initializationMode$ indicates the type of initialization. Also, a set of initial centers can be introduced using the $initialModel$ option. Finally, the number of steps to be executed during the K-means++ phase, $initializationSteps$, and a pre-defined error threshold for convergence, ϵ, should also be specified.

The operations performed by K-means|| in Apache Spark can be summarized as follows. First, the data are distributed among the slave nodes. Then, the master node prepares the algorithm environment, while the slave nodes run in parallel the for loop in line 4 of Algorithm 1 over its data portion. In this loop, $2k$ points are sampled on average for each run, with a probability proportional to their squared distance from that run's centers. When the loop is finished, more than k candidate centers have been calculated for each run. Each candidate center is weighed by the number of points in the dataset mapped to it. Finally, the algorithm runs a local K-means++ on the weighted centers to pick just k of them.

4.2 Multicore implementation on Scikit-Learn

Scikit is a Python Library for machine learning. This library contains its own K-means++ version, as an option of the $k-means$ class into the *sklearn.cluster* package, allowing the multicore execution of K-means method with the parameter n_jobs. Giving this parameter a positive value uses a number of processors equal to n_jobs. A value of -1 uses all available processors, while -2 uses one less, and so on. The default value of this parameter is 1.

[6] Since the K-means may not find the optimal overall solution, it is recommended to run it several times to converge to a better final solution. So, if $runs > 1$, for each iteration the total number of sets with different centroids that will be executed equals the number of $runs$.

The parameters of the K-means++ algorithm in Scikit are very similar to those in K-means||, since it is necessary to indicate the number of clusters, $n\_clusters$; the number of iterations of one complete execution of the algorithm, max_iter; and the number of times the K-means process will run with different centers, n_init.[7] Much like the Apache Spark version, Scikit's implementation has the option to initialize the first set of centroids randomly, using K-means++ or through an array of data, called $init$. Finally, it is necessary to indicate the relative increment (tol) in the results before declaring convergence, and the number of jobs that will be used during the execution, n_jobs. Each one of the jobs will execute a number of n_init runs in parallel (the number of total n_init runs will be distributed among the available jobs), thus there will be no speedup if a single run of the K-means algorithm is executed in the multicore implementation.

5 Experimental validation

5.1 Hyperspectral datasets

For the consistent experimental validation of the aforementioned implementations, we used two hyperspectral scenes from the well-known Indian Pines dataset,[8] acquired by AVIRIS [4] in 1992 over an agricultural site composed of agricultural fields with regular geometry and multiple crops:

1. The first scene has a size of 145 × 145 pixels, and it was collected over a mixed forest and agricultural area. It has 220 spectral bands in the range from 400 to 2500 nm, with spectral resolution of 10 nm, moderate spatial resolution of 20 nm, and 16 bits radiometric resolution. After an initial analysis, eight bands have been removed due to noise, ending up with a total of 212 bands. About half of the pixels in the image (10,366 of 21,025) have ground-truth information associated, which comes in the form of a single label assignment having a total of 16 ground-truth classes.
2. The second scene has a much larger size of 2678 × 614 pixels. It was collected over the same area, but spanning a much larger extent. It contains 220 spectral bands in the range from 400 to 2500 nm, with spectral resolution of 10 nm, moderate spatial resolution of 20 nm and 16 bits of radiometric resolution. The percentage of pixels with ground-truth information is about 20 % (334245 of 1644292) and the total number of classes is 58.

5.2 Hardware and software configuration

To evaluate the performance of both implementations, the underlying hardware architecture is composed of Intel(R) Xeon(R) CPUs E5430 @ 2.66GHz (8 cores), 16 GB RAM, Shared storage, NetApp FAS3140.

[7] After iterating, the algorithm takes only the best solution reached.

[8] Available online: https://engineering.purdue.edu/~biehl/MultiSpec/hyperspectral.html.

The distributed environment in which we have tested our cloud implementation is the one described in Sect. 2. As we have mentioned before, we have used the OpenStack over the hardware architecture as the cloud computing platform for experimental evaluation. This software is a collection of Open Source technologies that provide a scalable deployment of a cloud computing environment. Above this software we have implemented virtual nodes that have two VCPUs with 4GB of RAM and 40 GB hard disk each.

To establish a fair comparison between multicore and distributed version, we have also virtualized the hardware for the multicore experiments exactly with the same features as the ones used in Apache Spark. Therefore, the multicore algorithm has been executed on Intel(R) Xeon(R) CPUs E5430 @ 2.66GHz (8 cores) with shared storage NetApp FAS3140. It should be noted that this version needs 4GB of additional RAM due to particular storage requirements of the implementation. That means that we need 8GB of RAM in each CPU. On the software side, we used Java 1.8.0_92-b14, Ubuntu 14.04 $\times$64 LTS as operating system, Python 2.7.10, the newest (Scikit-Learn 0.18.dev0) version and the active development branch 1.6.2 of Apache Spark in our experiments.

5.3 Multicore and distributed algorithm configurations

Our first experiment is dedicated to evaluate the performance of the multicore implementation of the K-means++ algorithm developed with the Python Library, Scikit-learn. For each of the considered hyperspectral datasets, we run the K-means++ algorithm increasing the number of processing cores (2, 4, 6 and 8 cores), and repeat each test five times. The number of clusters is set to 16 different groups for the small Indian Pines image, and to 58 for the large Indian Pines image, with the tolerance threshold set to 1e $-$ 15. We consider ten different sets of centroids in the initialization process, and K-means++ iterates a maximum of 50 times over them. As already mentioned in Sect. 4.2, Scikit implements a multicore K-means version. If we want to execute the K-means++ variant, we must indicate it with the *init={'k-means++', 'random' or an array}* parameter. Moreover, this algorithm parallelizes using multiprocessing by default only when we introduce the parameter n_job. With $n\_job = 1$, no parallel computing code is used at all (only one job will be launched). However, if n_jobs is set below -1, $(n\_cpus + 1 + n\_jobs)$ CPUs are used. Our $n\_init = 10$ different centroid initializations are distributed among the available number of cores).

Our second experiment is focused on evaluating the performance of the distributed implementation K-means|| in Apache Spark. The execution of this implementation has exactly the same configuration as the multicore version: we have launched the K-means|| algorithm changing the number of computing nodes (1, 2, 3 and 4, with 2 virtual cores or VCPUs each). The number of clusters is set to 16 and 58, depending on the considered dataset, and the tolerance threshold is set to 1e $-$ 15. As in the previous version, K-means|| iterates a maximum of 50 times over the set of centroids. Apache Spark architecture needs a master node and several slave nodes (although there is no need that the master and the slaves execute physically in the same machine). In our experiments, we have one machine for each role. In all cases, we used one master node

and: (1) 1 slave node with 1 virtual core (to calculate the speedup); (2) 1 slave node with 2 virtual cores; (3) 2 slave nodes with 4 virtual cores in total; (4) 3 slave nodes with 6 virtual cores in total, and (5) 4 slave nodes with 8 virtual cores in total.

5.4 Performance evaluation

5.4.1 Clustering accuracy assessment

In this section, our goal is to highlight that both implementations (multicore and distributed) provide comparable clustering results with regards to the serial algorithm. We analyze the clustering results using two standard metrics: purity [7] and entropy [3].

- $Purity$ is calculated as the ratio between the number of correctly classified samples and the total samples, n, as follows: $purity(\omega, \mathbb{C}) = \frac{1}{n} \sum_k \max_j |w_k \cap c_j|$, where $\omega = w_1, w_2, \ldots, w_k$ is the set of clusters and $\mathbb{C} = c_1, c_2, \ldots, c_j$ is the set of classes. Perfect clusterings have a $purity$ of 1.
- $Entropy$ gives a measure of disorder (or uncertainty) of the clustering. It is calculated as follows: $H(\omega) = -\sum_{k=0}^{n-1} p(w_k) \cdot \log_2(p(w_k))$, where $p(w_k)$ is the probability of a pixel belonging to cluster ω_k. $Entropy$ is maximized ($H(\omega) = 1$) when all $p(w_k)$ have the same value, indicating that uncertainty is largest; and minimized ($H(\omega) = 0$) when $p(w_k) = 1$ and $\sum_{i=0, i \neq k}^{n-1} p(w_i) = 0$, when there is absolute certainty.

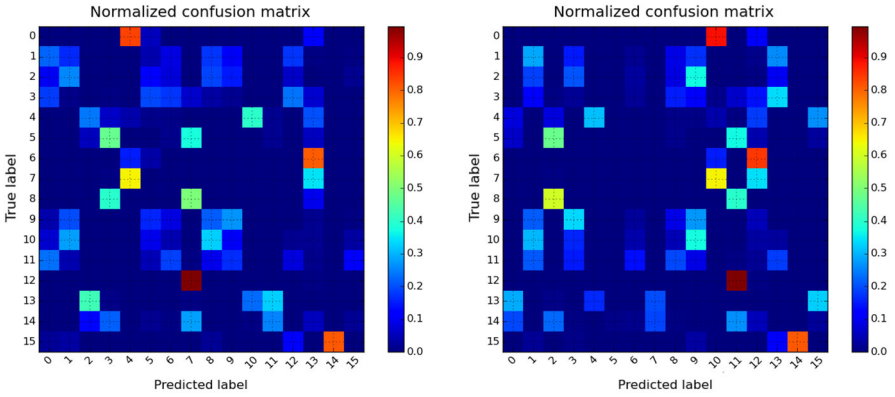
Table 1 shows the $purity$ and $entropy$ scores for the clustering results obtained by the multicore and the distributed implementations (note that for the experiments with the large image in the multicore, some of the clustering results could not be computed due to excessive memory requirements). The table reveals that both implementations provide almost exactly the same results for all executions. In addition, we use confusion matrices [14] to evaluate the agreement between the ground-truth classes and the clusters identified in the process (see Figs. 3, 4). Finally, we also show the clustering results obtained by the K-means++ and K-means|| algorithms for the small and large Indian Pines images (Figs. 5, 6) for visual inspection. Both figures show the clusters with the background of the image removed (i.e., those pixels that do not have associated ground-truth) and also without removing the background. Although the color assignment is different in the ground-truth and in the clustering results, the consistency of the clusters (obtained in fully unsupervised fashion) can be visually appreciated.

5.4.2 Performance of the multicore implementation

This section analyzes the execution time and speedup that can be achieved by the multicore implementation of K-means++. The results obtained by the Scikit implementation are summarized in Table 2, which shows the average and standard deviation for the execution time across the different repetitions of each execution of K-means++ over the two considered datasets. To calculate the speedup of the algorithm, we have also executed it using one single core. Table 2 shows how the execution time is reduced

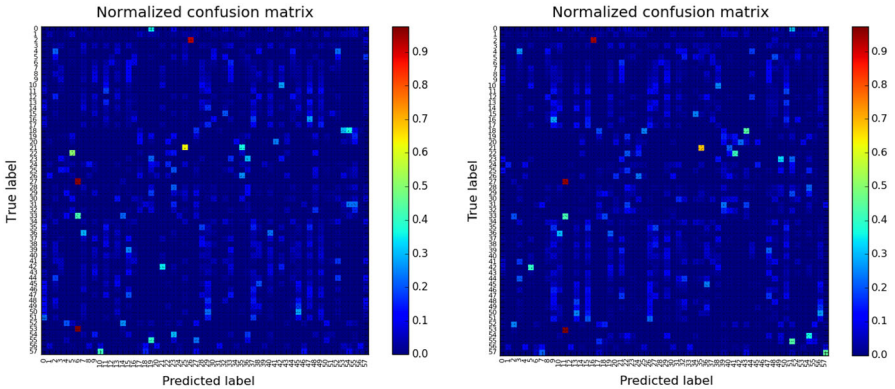
Table 1 Purity and entropy scores for the clustering results obtained by the multicore and distributed implementations (n.a. means not available due to excessive memory requirements)

Hyperspectral dataset	Small Indian Pines image				Large Indian Pines image			
	Multicore		Distributed		Multicore		Distributed	
Cores/nodes (cores)	Purity	Entropy	Purity	Entropy	Purity	Entropy	Purity	Entropy
1/1 (1)	0.5080	0.4602	0.5080	0.4608	0.3563	0.6054	0.3560	0.6091
2/1 (2)	0.5080	0.4609	0.5087	0.4703	0.3542	0.6003	0.3539	0.6029
4/2 (4)	0.5085	0.4644	0.5084	0.4602	0.3544	0.6041	0.3565	0.6029
6/3 (6)	0.5081	0.4602	0.5090	0.4678	n.a.	n.a.	0.3570	0.6014
8/4 (8)	0.5090	0.4656	0.5080	0.4600	n.a.	n.a.	0.3555	0.6100



(a) (b)

Fig. 3 Confusion matrix obtained after applying the K-means++ (**a**) and the K-means|| (**b**) algorithms to the small Indian Pines image **a** multicore execution, **b** distributed execution



(a) (b)

Fig. 4 Confusion matrix obtained after applying the K-means++ (**a**) and the K-means|| (**b**) algorithms to the large Indian Pines image **a** multicore execution, **b** distributed execution

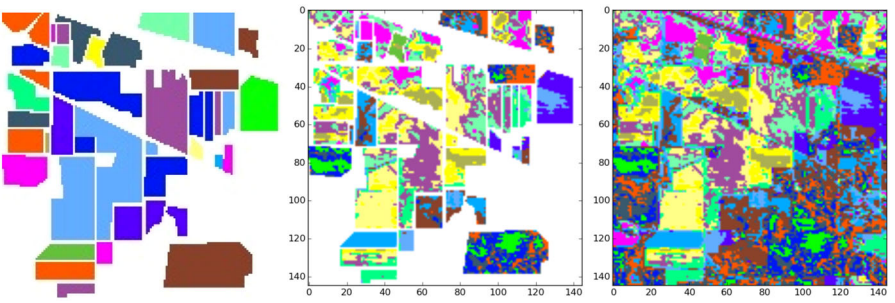


Fig. 5 Example of the clustering results obtained by the distributed K-means implementation over the small Indian Pines image. From left to right, we show the ground-truth information, the clustering result (*without background*) and the clustering result (*with background*)

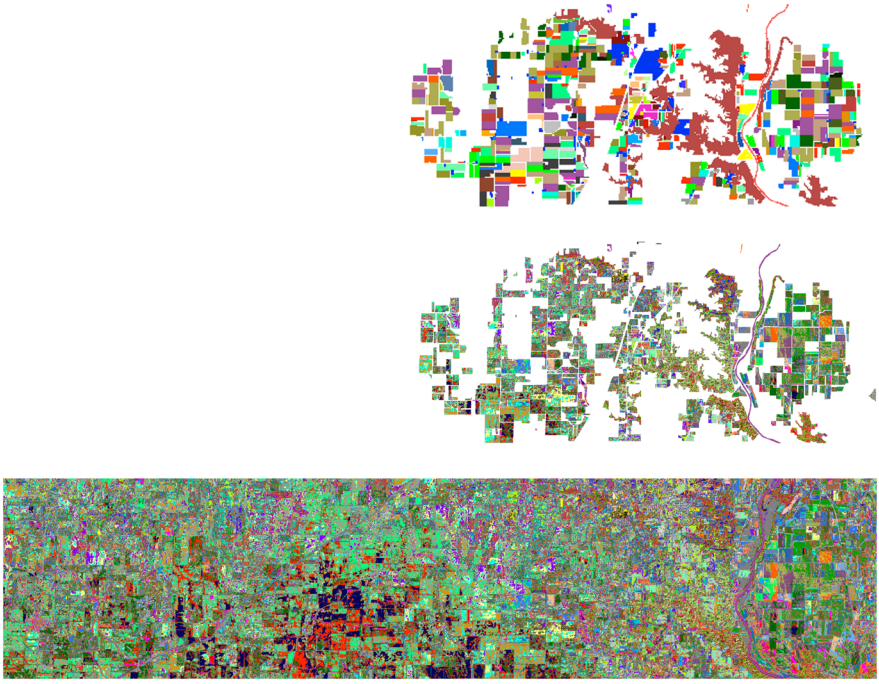


Fig. 6 Example of the clustering results obtained by the distributed K-means implementation over the large Indian Pines image. From top to bottom, we show the ground-truth information for the scene, the clustering result (*without background*) and the clustering result (*with background*)

by increasing the number of computation cores, obtaining an speedup of almost 2.2× when processing the small dataset with all eight Lavailable cores. It can also be seen in Fig. 7 that we obtain an acceleration factor of about 2.15× when processing the large datasets using four computation cores. The experimental configuration of the hardware does not allow the execution of the K-means++ over the large dataset using 6 or 8 cores due to excessive memory requirements.

5.4.3 Performance of the distributed implementation

This section focuses on the evaluation of the execution time and speedup that can be achieved by the distributed implementation of K-means||. The results obtained by the Apache Spark implementation are summarized in Table 3. As in Table 2, we show the average and standard deviation for the execution time across different repetitions of each execution of K-means|| over the two considered datasets.

As in the previous experiment, to calculate the speedup of the distributed version the algorithm has been executed with a single slave node, which has a single core. As we increase the number of machines, we can see in Fig. 8 how the speedup grows, reaching almost twice as fast with the small Indian Pines and almost six times faster with the large Indian Pines scene, using 4 nodes (8 virtual cores). We can clearly see how the algorithm scales better with large datasets. The K-means|| implementation

Table 2 Average execution time (in s) and speedup factor measured for each of the multicore execution of K-means++ with Scikit using both images and increasing number of available computing cores (n.a. means not available due to excessive memory requirements)

Hyperspectral dataset	Small Indian Pines image			Large Indian Pines image		
Number of cores	AVG Exec. time	Std Exec. time	Speedup	AVG Exec. time	Std Exec. time	Speedup
1	9.2549	0.2768	1	740.0427	1.0432	1
2	5.9628	0.1123	1.5521	445.5925	3.4084	1.661
4	4.8239	0.1152	1.9386	345.0600	1.0715	2.1447
6	4.3742	0.0708	2.1158	n.a.	n.a.	n.a.
8	4.2335	0.0682	2.1861	n.a.	n.a.	n.a.

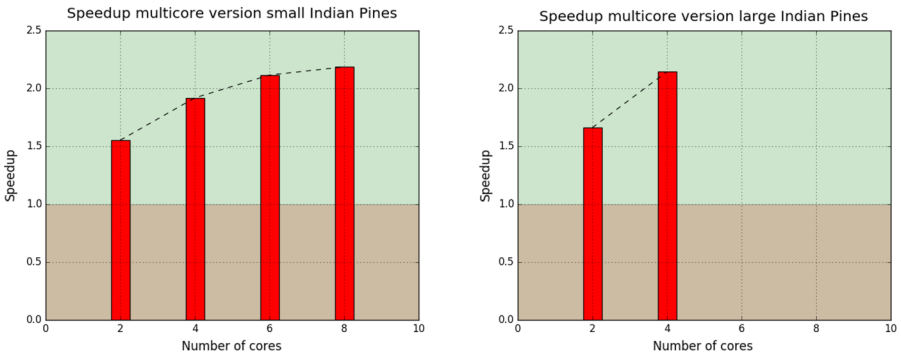


Fig. 7 Graphical representation of the K-means multicore implementation speedup with the Indian Pines images. For the large Indian Pines image, the execution using 6 or 8 cores was not possible due to excessive memory requirements **a** speedup small Indian Pines, **b** speedup large Indian Pines

Table 3 Average execution time (in s) and speedup factor measured for each of the distributed execution of K-means|| with Spark Apache using both images and increasing number of available computing nodes (the number of total cores is also shown in brackets)

Hyperspectral dataset	Small Indian Pines image			Large Indian Pines image		
Number of nodes (cores)	AVG Exec. time	Std Exec. time	Speedup	AVG Exec. time	Std Exec. time	Speedup
1 (1)	22.7896	0.2885	1.0	1239.7509	52.4981	1.0
1 (2)	13.9397	0.3071	1.6349	785.4585	36.9667	1.5784
2 (4)	12.9756	0.2476	1.7563	385.3718	9.0872	3.2170
3 (6)	12.9973	0.1441	1.7534	264.3419	6.7339	4.6900
4 (8)	13.0955	0.6274	1.7403	210.7660	10.3756	5.8821

uses both data-level and task-level parallelism, thus it is able of taking more advantage of the available hardware capabilities. In addition, the memory requirements are much more relaxed than in the multicore implementation.

6 Conclusions and future lines

In this paper, we have discussed the possibility of exploiting cloud computing architectures for hyperspectral image processing. As a case study, we have presented a cloud computing implementation of the popular K-means, an unsupervised clustering algorithm which is commonly used in the hyperspectral image analysis community. Our experimental results show the effectiveness of the proposed distributed implementation, not only in terms of clustering accuracy (comparable to the serial and multicore implementations of the algorithm) but also in terms of computational performance, particularly in the case of large hyperspectral datasets, for which we can fully exploit its parallel scheme. In addition, the distributed implementation is able to greatly alleviate

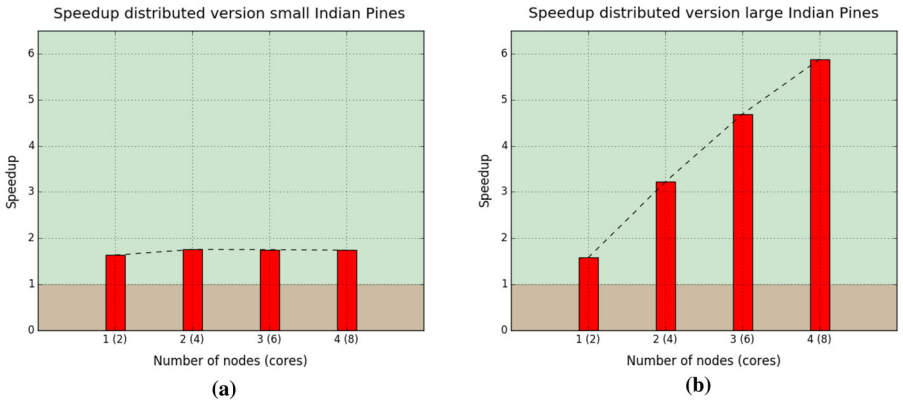


Fig. 8 Graphical representation of the K-means distributed implementation speedup for execution with the Indian Pines images **a** speedup small Indian Pines, **b** speedup large Indian Pines

the significant memory requirements of the serial and multicore versions. As a result, the distributed implementation of K-means is more amenable to be applied to large hyperspectral data repositories, while the multicore implementation can achieve better performance in the case of small datasets. As future work, we will implement other distributed algorithms for hyperspectral data processing with the aim of improving their performance.

Acknowledgements The authors would like to take this opportunity to gratefully thank the Editors and Anonymous Reviewers for their outstanding comments and suggestions, which greatly helped us improve the technical quality and presentation of the manuscript. This work has been supported by the Spanish Ministry of Science and Education (FPU grants). This work has also been supported by Junta de Extremadura (GR15005 grant). We acknowledge the use of the computing facilities at Extremadura Research Centre for Advanced Technologies (CETA-CIEMAT), funded by the European Regional Development Fund (ERDF), and particularly the system administrators Abel Francisco Paz Gallardo and Alfonso Pardo Diaz.

References

1. Arthur D, Vassilvitskii S (2007) K-means++: The Advantages of Careful Seeding. In: ACM (ed.) Eighteenth Annual ACM-SIAM Symposium on Discrete Algorithms, pp. 1027–1035. Society for Industrial and Applied Mathematics, New Orleans, Louisiana . 1283494
2. Bahmani B, Moseley B, Vattani A, Kumar R, Vassilvitskii S (2012) Scalable K-means++. Proc VLDB Endow (PVLDB) 5(7):622–633
3. Chang CI (2003) Hyperspectral imaging: techniques for spectral detection and classification. Kluwer Academic/Plenum Publishers, New York
4. Green RO, Eastwood ML, Sarture CM, Chrien TG, Aronsson M, Chippendale BJ, Faust JA, Pavri BE, Chovit CJ, Solis M, Olah MR, Williams O (1998) Imaging spectroscopy and the airborne visible/infrared imaging spectrometer (AVIRIS). Remote Sens Environ 65:227–248
5. Kanungo T, Mount DM, Netanyahu NS, Piatko CD, Silverman R, Wu AY (2002) An efficient k-means clustering algorithm: analysis and implementation. IEEE Trans Pattern Anal Mach Intell 24(7):881–892
6. León G, Molero JM, Garzón EM, García I, Plaza A, Quintana-Ortí ES (2015) Exploring the performance-power-energy balance of low-power multicore and manycore architectures for anomaly detection in remote sensing. J Supercomput 71(5):1893–1906

7. Manning CD, Raghavan P, Schtze H (2008) Introduction to information retrieval. Cambridge University Press, New York
8. Martínez JA, Garzón EM, Plaza A, García I (2011) Automatic tuning of iterative computation on heterogeneous multiprocessors with adithe. J Supercomput 58(2):151–159
9. Molero JM, Paz A, Garzón EM, Martínez JA, Plaza A, García I (2011) Fast anomaly detection in hyperspectral images with rx method on heterogeneous clusters. J Supercomput 58(3):411–419
10. Plaza A, Plaza J, Martin G, Sanchez S (2011) Hyperspectral data processing algorithms. In: Prasad AH, Thenkabail S, John G. Lyon (ed.) Hyperspectral remote sensing of vegetation, chap. 5, Taylor and Francis, Abingdon, United Kingdom, pp 121–137
11. Plaza A, Plaza J, Paz A, Sanchez S (2011) Parallel hyperspectral image and signal processing. IEEE Signal Process Mag 28:196–218
12. Plaza A, Plaza J, Valencia D (2007) Impact of platform heterogeneity on the design of parallel algorithms for morphological processing of high-dimensional image data. J Supercomput 40(1):81–107
13. Sevilla J, Bernabe S, Plaza A (2014) Unmixing-based content retrieval system for remotely sensed hyperspectral imagery on gpus. J Supercomput 70(2):588–599
14. Stehman SV (1997) Selecting and interpreting measures of thematic classification accuracy. Remote Sens Environ 62(1):77–89
15. Wu Z, Li Y, Plaza A, Li J, Xiao F, Wei Z (2016) Parallel and distributed dimensionality reduction of hyperspectral data on cloud computing architectures. IEEE J Sel Topics Appl Earth Obs Remote Sens 9(6):2270–2278