In-Memory Parallel Processing of Massive Remotely Sensed Data Using an Apache Spark on Hadoop YARN Model

Wei Huang, Lingkui Meng, Dongying Zhang, and Wen Zhang

***Abstract*—MapReduce has been widely used in Hadoop for parallel processing larger-scale data for the last decade. However, remote-sensing (RS) algorithms based on the programming model are trapped in dense disk I/O operations and unconstrained network communication, and thus inappropriate for timely processing and analyzing massive, heterogeneous RS data. In this paper, a novel in-memory computing framework called Apache Spark (Spark) is introduced. Through its merits of transferring transformation to in-memory datasets of Spark, the shortages are eliminated. To facilitate implementation and assure high performance of Spark-based algorithms in a complex cloud computing environment, a strip-oriented parallel programming model is proposed. By incorporating strips of RS data with resilient distributed datasets (RDDs) of Spark, all-level parallel RS algorithms can be easily expressed with coarse-grained transformation primitives and BitTorrent-enabled broadcast variables. Additionally, a generic image partition method for Spark-based RS algorithms to efficiently generate differentiable key/value strips from a Hadoop distributed file system (HDFS) is implemented for concealing the heterogeneousness of RS data. Data-intensive multitasking algorithms and iteration-intensive algorithms were evaluated on a Hadoop yet another resource negotiator (YARN) platform. Experiments indicated that our Spark-based parallel algorithms are of great efficiency, a multitasking algorithm took less than 4 h to process more than half a terabyte of RS data on a small YARN cluster, and 9*9 convolution operations against a 909-MB image took less than 260 s. Further, the efficiency of iteration-intensive algorithms is insensitive to image size.**

***Index Terms*—Apache Spark, big data, Hadoop yet another resource negotiator (YARN), parallel processing, remote sensing (RS).**

I. Introduction

In recent years, an ever-increasing volume of satellite remote-sensing (RS) data shows significant potential for scientists to explore global changes in patterns of atmosphere, soil, and water cycle of the earth. Satellites provide the big picture [1]. However, the huge volume of high spatial–temporal resolution RS data cannot be handled by a single computing node. The velocity of data production has overloaded current processor capacities. The variety of RS data means that image processing algorithms cannot be made generic for further analysis. Faced with such RS "big data" challenges [2], use of a parallel programming model such as MapReduce is needed for global research. Current work concentrates on special algorithms, and models demonstrate limited value in accelerating the knowledge-discovering process [3]–[5]. Very few operational projects involve how to parallel process massive RS data with generic programming models [6]. Practical cases that reveal performance gains on corresponding platforms are rather scarce [7], [8]. To this end, we here answer the basic question of how to design generic parallel algorithms that can be efficiently executed on candidate platforms for this work.

Regarding supporting platforms, high-performance clusters (HPC) and grid computing platforms (GCP) are not well suited for processing RS big data. The former (HPC) is powerful infrastructures filled with graphical processor units (GPUs) but with little runtime scalability. The latter (GCP) is a resource-sharing-oriented architecture but with little consideration of reliable resource provision [9], [10]. Moreover, it is important that MapReduce-like approaches are made available through service-oriented programming where the details have been wrapped behind the service interface [11]. A cloud computing model has shown great potential for ubiquitous, convenient, and on-demand network access to dynamic computing resources [12]. Therefore, cloud Hadoop has been suggested by many researchers for rapid RS processing and analysis [13]–[16]. However, cloud Hadoop is inappropriate for RS big data analysis because MapReduce only weakly supports multitasking and iterative-intensive RS algorithms [17], [18]. MapReduce-based algorithms suffer significant performance loss in dense disk I/O operations and unconstrained network communications, which have been widely reported [19]–[21]. Nonetheless, cloud Hadoop is the most widely used big data platform, in which the Hadoop distributed file system (HDFS) provides big data storage with high read throughput [22]. Most recently, an innovative framework called Apache Spark (Spark) has been open-sourced by University of California, Berkeley to cultivate the paradigm of in-memory computing and analyzing big data [23]. Berkeley proposed using main memory rather disk memory in MapReduce stages given that main memory channels have higher bandwidth than disks and other PCI devices. There is a general-purpose and scalable in-memory computing framework

Manuscript received December 28, 2015; revised Feburary 29, 2016; accepted March 21, 2016. This work was supported in part by the National High-Resolution Earth Observation System Projects (civil part): Hydrological monitoring system by high spatial resolution remote sensing image (first phase) (08-Y30B07-9001-13/15) and in part by the Fundamental Research Funds for the Central Universities under Grant 2014213020202. *(Corresponding author: Wen Zhang.)*

The authors are with the School of Remote Sensing and Information Engineering, Wuhan University, Wuhan 430079, China (e-mail: wen_zhang@whu.edu.cn).

Color versions of one or more of the figures in this paper are available online at http://ieeexplore.ieee.org.

Digital Object Identifier 10.1109/JSTARS.2016.2547020

for processing very large volumes of data at extremely high speed (e.g., a 100-TB sort in 23 min on 207 nodes in an Amazon.com cloud environment) [24]. This computing framework has also been implemented on a Hadoop platform. There is evident great potential to use a Spark on Hadoop platform for real-time processing large-scale or massive RS data. However, how we can generically design parallel RS algorithms, and what the performance gain is, remain unexplored.

In this context, Spark is introduced, with a focus on parallel processing massive RS data in Hadoop for the first time. The particular platform, Hadoop yet another resource negotiator (YARN) [25], known as the next generation of Hadoop, is proposed to provide the framework with YARN containers for concealing heterogeneousness of computing nodes. Compromising the fact that the variety and volume of RS data create great challenges in data partitioning policies and parallel programming difficulties in underlying computing nodes, we propose to use the strip-oriented parallel programming model that incorporates strip abstraction of RS data and resilient distributed datasets (RDDs) [26] for facilitating design of generic parallel RS algorithms. Strips are rather similar to rows of a table in a database and are key/value records. We use the unique keys of strips to localize pixels in a row of an image so that the Spark engine would dispatch transformation functions to nodes where these pixels reside. To better support Spark-based RS algorithms that process massive RS data stored in HDFS on a YARN platform, we implement a generic data partition method. RS data are logically partitioned into differentiable strips in the keys of which the image metadata are wrapped. We evaluate in detail both the availability and efficiency of the strip-oriented programming model employing a case that processes more than half a terabyte of RS data in a rather small cluster. The efficiency of pixel-level parallel processing such as widely used convolution and clustering operations in RS algorithms is also evaluated. Through the merits of Spark use on a YARN model, the proposed programming model and generic image partition method eliminate the need for parallel processing RS data with MapReduce and provide a promising way to achieve in-memory analysis of RS big data.

The rest of this paper is organized as follows. The Hadoop YARN platform and Apache Spark are briefly introduced in Section II, after which implementation of a generic parallel programming model and universal logical image partition method are described in Section III. In Section IV, the experimental environment and results of the case and algorithms use are introduced, followed with a discussion in Section V. Finally, conclusion and future work are introduced in Section VI.

II. Using Spark on Hadoop YARN

Hadoop is the open-source implementation of Google GFS [27] and MapReduce [28] for storage and parallel processing large volumes of data in a distributed wide-area network. With more than 10 years of development, Hadoop has become the most widely used big data platform. By separating the responsibilities of global resource management and job scheduling of MapReduce, its successor Hadoop YARN has become a standard big data operating platform. As shown in Fig. 1, ResourceManager, one of the core components of

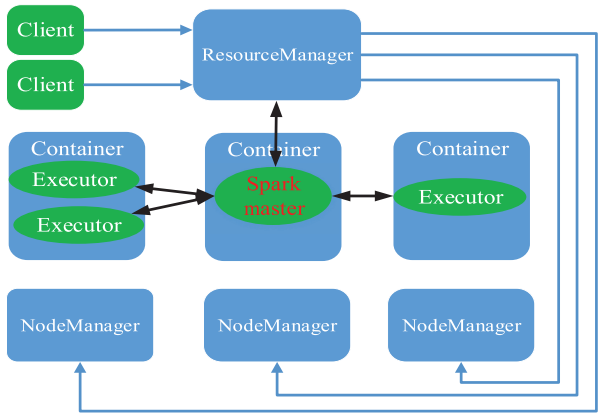

Fig. 1. Architecture of Spark YARN model. Blue rectangles represent core components of YARN, in which ResourceManager is responsible for accepting client requests and leasing containers to clients for a time period. NodeManagers distributed on multiple computing nodes are waiting commands from ResourceManager and responsible for monitoring the status of containers, and dynamically recycling containers on behalf of ResourceManager. Colored ellipses depict core components of Spark, in which the Spark master controls the execution of relevant tasks of applications and schedules tasks to Spark executors running on YARN containers. In particular, green ellipses represent Spark executors for executing concrete tasks of applications submitted by clients.

YARN, is responsible for managing and allocating containers to frameworks. Containers represent virtual cores and memory resources, which are hosted on cluster nodes. Node managers distributed on computing nodes will monitor the status of containers and execute administrate commands from ResourceManager to start up or recycle containers dynamically. It is the abstraction resource representation that makes Hadoop YARN conceal the heterogeneousness of the cluster environment for many well-known frameworks such as Apache Storm and Apache Tez [29], [30]. YARN delegates the tasks-scheduling of applications of these frameworks to each individual application master (AM), which makes decisions to schedule tasks to the allocated containers. Thus, it can conceal the variety of underlying computing nodes for these frameworks with high scalability. One may argue that the centralized resource manager should be a bottleneck as AMs may require large number of containers to process an unprecedented volume of data and AM itself has limited capacity to schedule large scale of tasks. Fortunately, many sophisticated works have been proposed to overcome the scaling limitations, which enable YARN to schedule arbitrary data processing applications in cloud [31]. Moreover, frameworks have the capacity to schedule tasks based on the data-locality, which can be valuable for processing RS big data. The obvious advantage also is used in this work.

A. Apache Spark

Spark adapts RDDs to represent distributed datasets partitioned across computing nodes. Each RDD is an abstraction of a dataset, which contains a number of partitions. Metadata of partitions location are also recorded in RDDs. Whenever a transformation is applied to an RDD, a new RDD will be generated. In the newly generated RDD, the transformation will be

recorded as a function object. The dependency is also recorded into the newly generated RDD object. Since many RDDs may be joined together to form a new RDD, a set of dependencies on the "Parent" RDDs would also be recorded into the new RDD object. The most frequently used transformation primitives are mapping, filtering, and joining, which represent simultaneously applying computation to all partitions of an RDD, filtering partitions of an RDD based on special criteria, and joining partitions from two RDDs, respectively [32]. Since a transformation to an RDD is very similar as declaring a pointer to an object, ideally, transformation functions can operate on all partitions locally without moving data blocks. Because a chain of transformation functions wrapped in RDDs can be used to restore the missing partitions of RDDs via recalculation on parent RDDs, the cost of the fault recovery mechanism known as lineage-based fault tolerance is constrained. Spark preferably uses memory to cache partitions. It is thus often called an in-memory computing framework.

It has different implementation of parallel processing phases as that of Hadoop. Each mapping task (M) directly writes the output to operating system's disk buffer and will create one shuffle file per reducing task. Unlike Hadoop, the output will not be merged into a single partition. Each reducing task (R) maintains a network buffer to fetch the outputs of mapping tasks. To mitigate shuffling pressure to the underlying operating system in creating $M*R$ number of files, Spark currently adapts shuffle file consolidation to reduce the number of shuffle files [33]. Note that M and R represent the number of mapping tasks and the number of reducing task, respectively. The results of mapping tasks that running on the same core (C) of processors of cluster will be written to only one consolidated file, which reduces the number of shuffle files to $C*R$. The underlying networking framework of Spark currently based on Netty, which makes the consolidated files can be directly copy to the socket without going the user-space memory for reducing CPU workload and memory pressure of Java virtual machines (JVMs) [34]. Spark shuffle phase is more efficient than that of Hadoop as reported in [35].

B. Spark on the YARN Model

Spark on YARN model works as follows: Spark master instance would first negotiate with YARN ResourceManager to obtain some containers when clients submit a Spark application to YARN platform. Spark master instance is the driver of the application, which is responsible for scheduling tasks to Spark executors running on the allocated containers. Spark master instance is the first process that would be loaded into a container. Containers would be dynamically allocated to Spark executors as requested by the Spark master instance. Those executors are thread pools and controlled by the Spark master instance for executing concrete tasks. Since the client can previously set specifications of driver and executors, YARN platform will meet the requirements according to the current workload and lease containers to the Spark application for a time period. After completing all of tasks, ResourceManager would revoke all the container resources leased to the Spark-based application.

From application views, assuming RS images stored in HDFS to be processed using Spark, Spark master will first create RDDs from HDFS, and then user-defined transformation functions on the RDDs will be mapped to parallel tasks that would be executed by Spark executors. Spark master would do all of the executing jobs such as scheduling tasks and intermediate result management. RDDs partitions would be cached in-memory preferably to speed up next transformations. User only need to define the transformation functions. Other than Spark on YARN model, Spark has other executing models such as Spark standalone model, Spark local model. The former means Spark will manage cluster resources on its own way, while the latter means Spark masters and executors are all running on a single computing node. We will evaluate the performance gain of Spark-based RS algorithms in this work. Since Spark on YARN is the most widely used model in the production environment, this work focus on Spark on YARN model particularly.

III. Generic Parallel RS Algorithms

Since RDD can only be created from stable storage such as HDFS or other RDDs, Spark-based algorithms that process RS data stored in HDFS suffer the impacts of image partition scale. Scale is not only related to efficiency in generating key/value records, but also the scale is highly correlated with algorithms for implementation and fault-tolerant cost recovery [36].

A. Image Partition Scale

Suppose that the total $\boldsymbol{N}$ megabytes of RS data are stored in HDFS and the default HDFS block size is $\boldsymbol{M}$ megabytes, and that the HDFS replication factor is denoted as $\boldsymbol{factor}$. Let disk I/O rate be $\boldsymbol{V}$ MB/s, the pixel number of a RS image be proportional to the size of the RS image, and each pixel be $\boldsymbol{I}$ bytes. Suppose that there are $\boldsymbol{H}$ hosts and each has $\boldsymbol{C}$ MB/s processing speed. Assuming scheduling costs are negligible, the total processing time T for the simplest operations such as inversion of the pixel digital number can be formulated as following:

$$T = \frac{\frac{N}{M} \times \left(\frac{M}{V} + \frac{2^{20} \times M}{IC} \right)}{factor \times HC} \tag{1}$$

where the numerator represents the time required to read all pixels from HDFS, and the denominator represents the sum of processing capacities of the underlying hardware. The two components in the bracket of the numerator represent time cost to read data, and time cost for key/value records generation, respectively. As shown in the above formula, the time cost of key/value records generation is unrestricted. The frequently used term "logical partition" in MapReduce-based RS algorithms actually adds a scale to the denominator as shown in the following formula. When partitioning to pixels, scale equals 1

$$T = \frac{\frac{N}{M} \times \left(\frac{M}{V} + \frac{2^{20} \times M}{Scale \times IC} \right)}{factor \times HC}. \tag{2}$$

Therefore, Spark-based RS algorithms must consider two factual aspects: if image partition scale is too small, time cost of key/value records generation will increase, and fault recovery cost will increase if the whole transformation chain needs to be recalculated. Note that the transformation chain preserved in RDDs may require re-execution of the initial key/value records generation operations. If the scale is too large, not only network communication cost will increase but also the probability of "out of memory" (OOM) errors will occur. Because units of data may traverse different HDFS blocks, remote fetching blocks from other nodes is inevitable [37]. JVMs have limited space. OOM errors occur if the size of image units overflows memory capacity of JVMs at computing nodes. This is why previous works concentrated on well-designed partition methods [38], [39].

Considering the fact that data partitioning to the pixel scale may help to implement pixel-level parallel algorithms, but would increase the time for necessary key/value pair generation, partitioning to the band scale may help to implement band-level parallel algorithms (BLPAs). However, this would increase the possibility of OOM errors, and we propose to use strips as logical image partition units. Strips are intermediate units of RS data and are a familiar concept to RS experts. In TIFF standard, strips are used for accelerating image processing and frequently represent rows of TIFF images. Our strip-programming model is based on defining rows-based RDDs from imagery. Because strips can be more efficiently localized and processed, we use key/value strips to conceal the variety of RS data in Spark-based RS algorithms. We have designed a generic partition method for Spark-based algorithms to extract strips from HDFS blocks by implementing two classes named RsInputFormat and RsRecordReader. The former class extends the "FileInputFormat" for logically splitting images into partitions with the default HDFS block size. The latter class implements a "RecordReader" interface for generating key/value strips in which RS metadata contents are wrapped in the keys [36]. To mitigate network communication, each RsRecordReader instance first judges whether each strip of an image is mostly contained in a HDFS block, and then reads the strip if the size exceeds 80%. Image metadata do not have to be separately stored and consume additional storage space. Because Spark adapts lazy computing only when RS data are processed, these strips would be generated then. Other widely used image formats such as GeoTIFF [40] and HDF-5 [41] have no terms "Strip," but we can still use strip abstraction to represent one row of images of these types. For example, we can use NetCDF library for reading content of the images [42], and then generate meaningful strips. As is known, RecordReader class is used to generate records for each input split. In this work, Strip-oriented processing model based on defining RDDs based rows from imagery.

We have proposed the generic image partition method based on the following reasons: first, strips are neither too big nor too small for computing nodes in a highly heterogeneous environment. Parallel computing units can use the keys to localize any part of an image rather efficiently. Spark-based RS algorithms using strip-oriented processing model have little chance to encounter OOM errors. Typically, an image often has limited number of columns that is small enough which makes its possible load them in distributed JVM memory. Second, the keys of the strips can be used to differentiate each other. Parallel RS algorithms are easy to implement. Since each strip corresponds to one row of an image, Spark-based algorithms can first localize rows where the pixel of interest resides, and then localize the desired columns. Third, RS image metadata content is typically small, thus wrapping them in the keys of strips does not impact the underlying shuffle system that uses hash functions to reorder the strips. Last, strip-oriented algorithms can be highly efficient; this is evaluated in Section IV.

We identified three categories of RS parallel algorithms that cover most of RS algorithms according to the classification method proposed by Ma *et al.* [43]. The first category is pixel-level parallel processing, which includes pixel-based processing and neighboring pixels processing. Pixel-based processing performs computations on individual pixels without reference to other pixels. Such algorithms include radiometric correction. Neighboring pixels processing refers to processing using pixels in the same position or their close neighbors of the same images. Resampling and convolution operations are of this kind. The second category is band-level parallel processing. Common algorithms include band math as such normalized difference vegetation index (NDVI) [44] and NDWI [45] that frequently used in the RS community. The third category is irregular parallel processing. This processing performs computations on irregular parts of pixels of an image; ISODATA [46] and K-means [47] are typical representations. Because irregular algorithms can be divided into multiple parallel stages, we do not differentiate this last type of parallel processing with multitask parallel processing in this paper. The rest of this section details how parallel RS algorithms can be easily expressed with our strip-oriented parallel programming model.

B. Generic Parallel RS Algorithms

Suppose that I1 and I2 be RS data that have the same columns and rows, and that f$(\cdot)$ represents the transformation function. Because the strips of an image are similar to rows in a database table, some parallel primitives used by Spark-based algorithms can be explained as follows.

$\boldsymbol{R} = \boldsymbol{join}(\boldsymbol{I}1, \boldsymbol{I}2)$ represents a joining operation that combines two images, if and only if two images have one too many of the same keys of strips. $\boldsymbol{R}$ represents the final image, of which the keys are identical with the keys of $\boldsymbol{I}1$.

$\boldsymbol{R} = \boldsymbol{map}(\boldsymbol{f}(\boldsymbol{I}1))$ represents a transformation function $\boldsymbol{f}$ () which will be applied to each strip of $\boldsymbol{I}1$; the function will change each strip. $\boldsymbol{R}$ represents the final image, of which the keys of strips may differ from those of $\boldsymbol{I}1$. Note that the inner function $\boldsymbol{f}$ () is the parameters of the outer ***map*** function.

RS raster images store their raw data based on only three particular formats known as band interleaved by pixel (BIP), band sequential format (BSQ), and band interleaved by line (BIL). Therefore, by incorporating strip abstraction, pixels and bands are localized irrespective of what the format of the image is. Note that we only focus on processing huge volume of raster data. Four dimensional RS data parallel processing is outside scope of this paper. For example, a multispectral RS image in

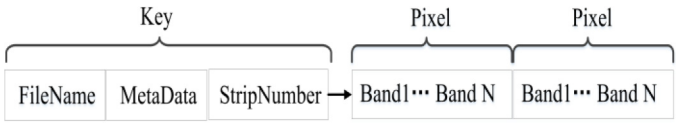

Fig. 2. Image of BIP format includes key/value pair strips that are generated by the Spark engine at runtime. The key part consists of file name, metadata, and strip number. The value part consists of multiple pixels and each pixel may consist of multiple bands.

BIP format has meaningful strips as shown in Fig. 2. Since pixel-level and BLPAs can be easily expressed as the combination of the parallel primitives, we do not illustrate the all-level parallel algorithms example. The following codes can be used to change pixels of an image according to the pixels of the same location from another image (assuming that the two images have the same resolution).

$\boldsymbol{R1} = \boldsymbol{map}\ (\boldsymbol{f_1}(\boldsymbol{I}1))$ // Change the keys of strips of image $\boldsymbol{I}1$ to form $\boldsymbol{R1}$.

$\boldsymbol{R2} = \boldsymbol{map}\ (\boldsymbol{f_2}(\boldsymbol{I}2))$ // Change the keys of strips of image $\boldsymbol{I}2$ to form $\boldsymbol{R2}$.

$\boldsymbol{R3} = \boldsymbol{join}\ (\boldsymbol{R1}, \boldsymbol{R2})$ // Combine $\boldsymbol{R1}$ and $\boldsymbol{R2}$ to form $\boldsymbol{R3}$.

$\boldsymbol{R} = \boldsymbol{map}\ (\boldsymbol{f}(\boldsymbol{R3}))$ // Apply function $\boldsymbol{f}()$ to each strip of $\boldsymbol{R3}$.

In this paper, we have used BitTorrent-enabled broadcast variables to share the intermediate results of employing multitasking RS algorithms. Broadcast variables were partitioned into many pieces by the Spark master. After subtasks of a multitasking RS algorithm were executed, the executor would use BitTorrent protocol to obtain shared data from both the Spark master and multiple Spark executors. The P2P-based data sharing mechanism will be rather useful for parallel tasks. The following pseudocode broadcasts a compressed image to the subtasks.

$\boldsymbol{R1} = \boldsymbol{map}$ (compress ($\mathbf{I1}$))// Compress the big image.

$\boldsymbol{R2} = \boldsymbol{broadcast}\ (\mathbf{R1})$ //Create a broadcast variable.

$\boldsymbol{R} = \boldsymbol{map}\ (\ f(\mathbf{I1}, \mathbf{R2}))$.

The last line changes the pixel's value of each strip of the image $\boldsymbol{I1}$ and uses the broadcast variable $\boldsymbol{R2}$ to access contents of the compressed image in function f (). Spark has another method called accumulator [30] to share data between parallel subtasks. Therefore, the method has little value to Spark-based RS algorithms. In this paper, we introduce the broadcast variables with strips for efficient sharing large volumes of RS data. Note that broadcast variables are not suitable for sharing large images. Regarding memory size of Spark driver, we can set the "Spark.driver.memory" parameter in the "Spark-defaults.conf" file to enlarge the memory size of the Spark driver and let the Spark engine choose a powerful node for the driver. However, there is no such convenient way to adjust the memory size of Spark executors. The difficulty here lies in two factors. The only way to adjust the maximum memory size of Spark executors is at submission time. When the "–executor-memory" parameter is too large, there would be fewer containers that can be leased by ResourceManager. If the parameter value is too small, there would be no more space for Spark executors to cache intermediate results and execute concrete tasks. Nonetheless, we can still use the key/value strips, and let the parallel subtasks to retrieve whatever part of the images through strips are without OOM errors. Therefore, strip abstraction has two roles in this context. First, it can facilitate parallel programming in a heterogeneous cloud without worrisome OOM errors. Second, it can be used to share large volumes of RS data valuable for data-intensive multitasking RS algorithms.

To conclude this section, we introduce a generic parallel model for parallel processing massive RS data in-memory in rather heterogeneous YARN clusters. By combing strips of RS data with RDDs, all-level parallel algorithms can be easily expressed with the primitives and the broadcast variables. The performance of all-level parallel algorithms will be evaluated in Section IV.

IV. Results and Analysis

A. Experiment Environment

The experiment environment consisted of nine nodes. We used 4 Dell PowerEdge M610 servers, with three of them having 12 CPUs, 48 GB main memory, and 160 GB disk, and one of them having 12 CPUs, 92 GB main memory, and 500 GB disk memory. Also, we used 1 Dell PowerEdge T630 main server with 12 CPUs, 32 GB main memory, and 3 TB disk, 3 HP Z220 workstations with each having 8 CPUs, 32 GB main memory, and 3 TB disk, and 1 Dell OptiPlex 990 PC with 4 CPUs, 28 GB main memory and 1 TB disk. We employed a Oracle Virtual Box 5.0.0 to create three virtual nodes, and two of them are hosted on two HP Z220 workstations, while one is hosted on the Dell PowerEdge T630 main server. To monitor cluster performance, a Ganglia monitor (core 3.6.0) is used [48]. The first three nodes in Table I adapt as OS Ubuntu 14.04.2 LTS, while the others adapt Ubuntu 12.04.5 LTS. The Hadoop version used is hadoop-2.4.0. The Spark version is Spark 1.4.0. The Scala interpreter version is Scala 2.10.4. The JDK version is JDK 1.7.0_45. The cluster is heterogeneous with different hardware (servers, workstations, and PCs), different networks (two subnetworks), and different operating systems (Windows and Linux). Computing nodes having better performance are configured with less disk space, while nodes with fewer CPUs are configured with more storage disks. To optimize the execution of Spark applications on a Hadoop YARN model, parameters were tuned according to Cloudera's Guide [49]. Specifically, we preserved the necessary memory space for OS and its daemon services using "yarn.nodemanager.resource.memory-mb" to limit the amount of physical memory, in MB, that can be allocated for containers. Also configured parameters are "yarn.scheduler.minimum-allocation-mb" and "yarn.app.mapreduce.am.resource.mb" which control the smallest amount of physical memory, in MB, that can be requested for a container and the physical memory requirement, in MB, that can be allocated for AMs, respectively. YARN container were set with at least 2 GB physical memory. AMs were configured with at least 4 GB memory by configured the two parameters in "yarn-site.xml" file. Many of parameters were tuned according to our tens of experiments, we expected that containers with 2 GB memory and AMs with 4 GB memory could satisfy many data-intensive algorithms. The total usable memory for applications of the

TABLE I
SPECIFICATION OF EACH CLUSTER NODE OF THE EXPERIMENT ENVIRONMENT

Node	Machine type	Specification	Actor
192.168.200.97	Physical machine (Dell PowerEdge M610 server)	12 CPUs, 48 GB memory, and 160 G disk	Hadoop master Hadoop slave Spark master Spark slave
192.168.200.109	Physical machine (Dell PowerEdge M610 server)	12 CPUs, 48 GB memory, and 160 G disk	Hadoop slave Spark slave
192.168.200.111	Physical machine (Dell PowerEdge server M610)	12 CPUs, 48 GB memory, and 160 G disk	Hadoop slave Spark slave
192.168.203.16	Physical machine (Dell PowerEdge M610 server)	12 CPUs, 92 GB memory, and 500 GB disk	Hadoop slave Spark slave
192.168.203.25	Physical machine (HPZ220 workstation)	8 CPUs, 32 GB memory, and 3 TB disk	Hadoop slave Spark slave
192.168.203.42	Physical machine (Dell OptiPlex 990 PC)	4 CPUs, 28GB memory, and 1 TB disk.	Hadoop slave Spark slave
192.168.203.47	Virtual machine (Hosted on HP Z220 workstation)	8 CPUs, 24 GB memory, and 2 TB disk	Hadoop slave Spark slave
192.168.203.48	Virtual machine (hosted on HP Z220 workstation)	8 CPUs, 24 GB memory, and 2 TB disk	Hadoop slave Spark slave
192.168.203.19	Virtual machine (hosted on Dell PowerEdge T630 main server)	12 CPUs, 24 GB memory, and 2 TB disk	Hadoop slave Spark slave

cluster was 342 GB. Other configurations such as JVM heap size were also considered. As shown in Table I, daemons of the Hadoop master the Spark master were running on the same nodes, with other nodes being used for storage and computing. RS data were previously uploaded to HDFS. The default block size of HDFS was 128 MB, and the replication factor equaled 3. The heterogeneous cluster was used to simulate the infrastructure used in data centers which consist of many commodity PCs.

B. Experiment Design

As discussed earlier, leasing more containers from ResourceManager can improve the efficiency of Spark-based RS algorithms. However, the investment then also can be higher. Thus, it is necessary to evaluate the performance gain using all cluster resources to process large volumes of RS data. Additionally, understanding the impact of the volume of RS data to be processed that exceeds the cluster memory is also necessary. Furthermore, multitasking algorithms that share large volumes of data need to be carefully investigated. Therefore, in Section IV-C, we designed a use case on Spark-based algorithms that include multiple data-intensive computing stages and will be used to process more than a half terabyte RS data. Because performance benchmark on pixel-level parallel can be used to analyze both the initialization cost of Spark-based algorithms and the efficiency of the proposed logical partition method, we designed a parallel algorithm that changes every pixel of RS data. Then, in Section IV-D, we paid attention to the time cost of more useful neighboring pixel processing parallel algorithms that employed four Spark execution models to evaluate the impact of the image size and convolution windows size on the kind of algorithms. Because the variety of RS data such as image size and storage format can influence the performance of BLPAs, we compared in Section IV-E the time cost of NDVI computing that processed two different datasets to obtain the main factors that influence the efficiency of BLPAs. Finally, we investigated the influence of the container number, image size, and degree of algorithm complexity on irregular-level parallel algorithms in Section IV-F. All of the Spark-based algorithms were designed with the proposed strip-oriented parallel programming model.

C. Availability of Spark on a YARN Model

In order to evaluate the availability of Spark on a YARN model, we chose three RS indices highly correlated with soil moisture for simulating soil moisture estimations of cropland fields at global scale using long time-serial MODIS imagery. The three RS indices are fractional vegetation cover (FVC) [50], land surface temperature (LST) [51], and land surface albedo (LSA) [52]. Soil moisture is key in controlling hydrothermal circulation between land surface and atmosphere through crop evaporation and transpiration [53]. Much global scale research needs to extract these indices from huge volumes of RS data in modeling global climate change patterns [54], [55]. Terra MODIS daily land surface reflectance (LSR) product (MOD09GA) [56], day LST product (MOD11A1) [57], and Global LAnd Surface Satellite daily shortwave (300–3000 nm) albedo product (GLASS02A01) [58], [59] on the 1st, 11th, 21st of May to October each year from 2000 to 2014 were used to calculate FVC, day valid LST, and ground true albedo. MODIS land cover type products (MCD12Q1) [60] were acquired for croplands mask extraction each year from 2001 to 2014. In order to fill in the gap with the 2000 MODIS land cover product, we substituted the global land cover map of the year 2000 (GLC2000) at 1 km resolution, distributed by the Joint Research Center in Italy for the product [61]. There were in total 497 GB of MOD09GA images, 115.5 GB of GLASS02A01 images, 37.4 GB of MOD11A1 images, and 2.62 GB of MCD12Q1 images. The number and size of the images were very different. There were 1060 MOD09GA images, with each having 16 800 columns and 12 000 lines; 532 MOD11A1 images, each with 7200 columns and 4800 lines; 1068 GLASS02A01 images each with 8400 columns and 6000 lines; and 14 MCD12Q1 images, each having 16 800 columns and 12 000 lines. Spark-based algorithms use all of

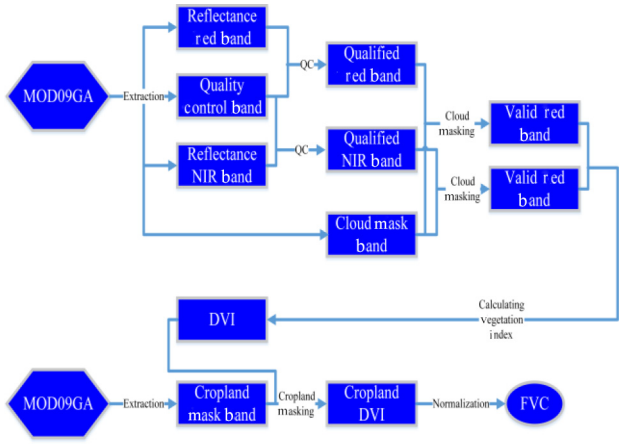

Fig. 3. Procedure of FVC calculating. It consists of a QC stage, a cloud masking stage, a VIC stage, a cropland masking stage, and a normalization stage. Hexagons represent the HDFS directory where images reside. Rectangles represent bands of the images as the inputs of the stages. The eclipse represents the final FVC indices.

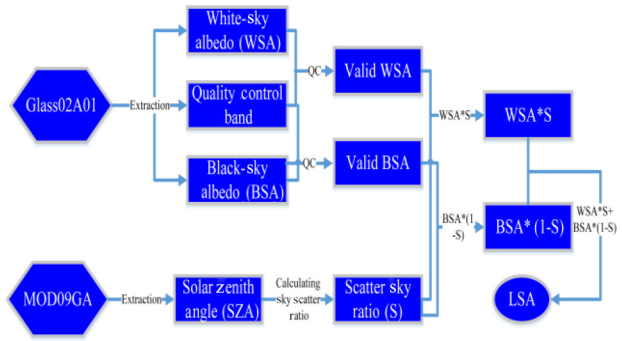

Fig. 4. Procedure of LSA calculating. It consists of a QC stage, a sky scatter ratio calculating stage, and a SAC stage. SAC consists three joining operations as following: valid WSA and valid BSA will join scatter sky ratio (S) to build BSA*(1-S) and WSA*S, respectively. And then they will be joined together to generate the final LSA indices depicted by the eclipse. Hexagons represent the HDFS directory where images reside. Rectangles represent bands of the images as the inputs of the stages.

Fig. 5. Procedure of calculating LST. It consists of a QC stage, a normalization stage. Hexagons represent the HDFS directory where images reside. Rectangles represent bands of the images as the inputs of the stages. The eclipse represents the final LST indices.

the cluster resources by setting command arguments with "–num-executors 170 –executor-memory 2g –driver-memory 2g." As shown in Figs. 3–5, FVC is the most complex algorithm that includes quality control (QC), vegetation index calculating (VIC), land cover mask (LCM), and normalization stages. LSA is the less complex algorithm that includes QC, scatter sky ration calculating (SSRC), and surface albedo calculating (SAC) stage. LST is the simplest algorithm and includes only QC and normalization stages. These algorithms include pixel-level parallel processing such as QC, band-level processing such as the difference vegetation index (DVI), and are composed of multitasking stages. Five performance metrics were selected to evaluate the efficiency of the algorithms. They were TE, CSE, MEAN, MIN, and MAX. Spark employs stage-oriented scheduling algorithms, in which tasks of each stage will be assigned to multiple Spark executors. We used TE to represent the total execution time of the algorithms. CSE indicates the time cost of some critical stages of these multitasking algorithms, and MEAN, MIN, and MAX are employed to measure the percentiles of task completion times in the critical stages.

TABLE II
TOTAL EXECUTION TIME OF FVC, LSA, AND LST

Algorithms	Data size (GB)	Image number	Image resolution	Total time	Execution
FVC	497	1060	16 800*12 000	3 h, 28 s	50 min,
LSA	115.5	1064	8400*6000	1 h, 51 min, 25 s	
LST	37.4	532	7200*4800	24 min, 46 s	

The three algorithms were executed 10 times, and the shortest runs were recorded. These algorithms obtained impressive overall performance gain. As shown in Table II, more than half a terabyte RS data that underwent complex transformation procedures was done in a few hours on the cluster. Even when data volume considerably exceeded the usable memory capacity of the cluster—each image of 1060 MOD09GA data has hundred millions of pixels—Spark in YARN model can perform fairly well. Using a 342 GB cluster memory to extract FVC indices from the 497 GB MOD09GA images took less than 4 h in the nine machine cluster. We noticed that there is a data sharing stage in the procedure of FVC calculating. In total, 14 years of MCD12Q1 images needed to be shared among tasks of the DVI stage to obtain cropland DVI indices as shown in Fig. 3. Thus, broadcast variables should be used to share 2.62 GB RS data in this context. Since the large volume of data needs to be collected by the Spark driver before being accessed by the tasks of cropland DVI stage, we measured the time cost of the collection phase many times outside of the tests. It took less than 2 min to collect the 2.62 GB RS data. The results show that it is suitable to share a large volume of data via our proposed method. TE of LSA was 1 h, 51 min, 25 s, and TE of LST was 24 min, 46 s. The results thus show that impressive performance gains can be obtained when memory can accommodate all of the data. The difference between TE of FVC and that of LSA can be explained as the in-memory computing advantage provided by the Spark engine. Since the 115.5-GB GLASS02A images could be entirely loaded into memory, LSA completed all its tasks in-memory. In contrast, for the 497 GB MOD09GA images, there were many operations in the Spark engine that evicted blocks OOM and reloaded them from disks. The procedure of LSA calculating includes multiple joining operations and is rather different from that of FVC calculating. First, black-sky albedo (BSA) and white-sky albedo (WSA) will join corresponding quality layers to form a valid

TABLE III
DETAIL OF CRITICAL STAGES OF FVC, LSA, AND LST

Algorithms	Critical stage	Execution time (min)	Task number	MIN (s)/MEAN (s) /MAX (s)
FVC	Quality Control	10.5	3975	0.4/7.5/35
	Vegetation Index Calculating	6.6	3975	0.5/8/33
	Cropland Mask	8.1	3975	0.6/7/32
	Normalization	3.7	3975	0.2/4/18
LSA	Quality Control	20.7	1230	6.5/57/358
	Sky Scatter Ration Calculating	6.1	1230	1/32/94
	Surface Albedo Computing	2.1	1230	0.3/36/66
LST	Quality Control	5.2	532	15/54/243
	Normalization	3.1	532	12/36/186

BSA layer and WSA layer, respectively. Then, the two valid layers will join scatter sky ratio (S) to build BSA*(1-S) and WSA*S, respectively. Finally, the two parts are joined together to form LSA indices. Therefore, five joining operations would be required to calculate LSA as shown in Fig. 4. As discussed earlier, more joining operations mean more shuffle operations have to be done by the Spark engine, because it has to reorder the key/value records in each of the stages. Nonetheless, LSA indices were obtained in less than 2 h. Furthermore, all of the critical stages completed in less than 21 min. The CSE of the QC stage of FVC was 10.5 min, CSE of the VIC stage of FVC was 6.6 min, CSE of LCM stage of FVC took 2.0 min, and CSE of SSRC stage of LSA required 6.1 min. These stages are data-intensive computing stages and often executed following joining stages. Since function objects would be serialized to the partitions of joined RDDs, all tasks of the stages were executed locally as expected. This is the main reason that many subtasks completed in a few seconds, down to even hundreds of milliseconds. More than 50% tasks of VIC cost less than 8 s as shown in Table III. These results show that Spark in a YARN model is suitable for both data-intensive and join-intensive RS algorithms. The strip-oriented parallel processing method is thus of high availability for RS algorithms. Irrespective of the fact that shuffle operations would comprise about 65%–97.5% of the runtime costs of the algorithms, all of the algorithms produced impressive performance gains. We also observed that task number of the critical stages generated by Spark engine will be determined by HDFS block size. One thousand and sixty MOD09GA images stored in HDFS were partitioned into $(497\ \text{GB}/128\ \text{MB} = 3975)$ *3 blocks in HDFS data nodes. The Spark engine would generate 3975 tasks to complete the FVC stages because the size of our logical image partitions was the same as of the HDFS block size. The design is rational since the same codes can be adapted to different YARN clusters without any modification.

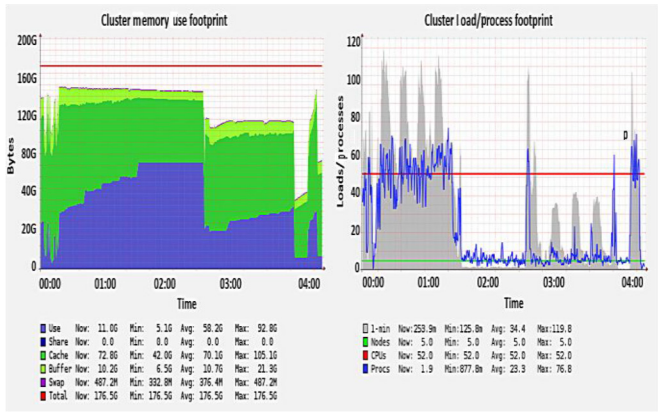

Fig. 6. Cluster memory usage and workload footprint of FVC calculating.

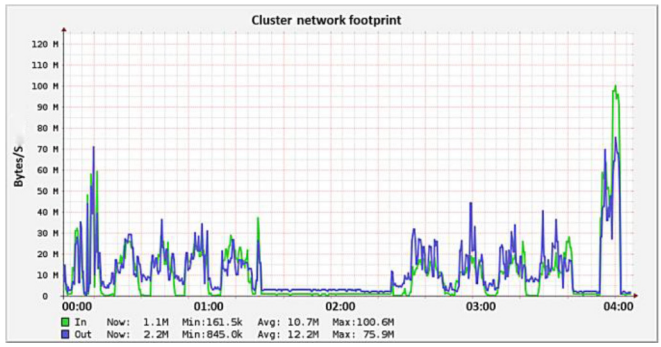

Fig. 7. Cluster network footprint of FVC calculation.

To observe the memory usage of Spark in a YARN model, we tracked the whole procedure of FVC calculating using a Ganglia monitor [48]. On average, there were 70.1 GB of memory used for caching data in the whole procedure. There were at most 105.1 GB of memory used to cache intermediate data. The results demonstrated that 20.4%–30.7% of memory would be used to cache data $(70.1\ \text{GB}/342\ \text{GB} = 0.204,\ 105.1\ \text{GB}/342\ \text{GB} = 0.307)$. Nearly 51.3% of memory would be employed to cache data and execute tasks $(175.5\ \text{GB}/342\ \text{GB} = 0.513)$. This is why many tasks completed in milliseconds as shown in Table III. The load/process footprints showed that even the data-intensive algorithms FVC cannot exceed the entire cluster computing capacity. An interesting phenomenon appeared in the procedure of FVC calculation. From time 0:00 to 1:00, there would be an obvious wave peak, while in the rest of time, there would be a wave trough as shown in Fig. 6. The wave peak indicated that there would be many data caching and shuffle operations occurring before actual RS indices calculating stages. The wave trough reflected that the calculating stages would be locally executed with limited network communication as shown in Fig. 7. The results showed that Spark-based algorithms employ distributed memory to cache data and have constrained networking communication.

All of above results show that the strip-oriented parallel programming method is suitable for data-intensive multitasking RS algorithms. Even when the volume of RS data exceeds the size of usable cluster memory, algorithms can obtain perform gains.

TABLE IV
BENCHMARK OF EMBARRASSED PIXEL-PARALLEL OF SPARK STANDALONE AND SPARK ON YARN MODEL IN THE CLUSTER

Executing model	Data size (GB)/ image numbers	Total strips	Elapsed time (s)
Spark on YARN	117.1/1456	6 988 800	1754.41
Spark on YARN	405.8 /864	10 368 000	3886.98
Spark on YARN	37.4 /532	3 192 000	356.52
Spark Standalone	117.1/1456	6 988 800	1110.47
Spark Standalone	405.8 /864	10 368 000	3590.04
Spark Standalone	37.4 /532	3 192 000	224.27

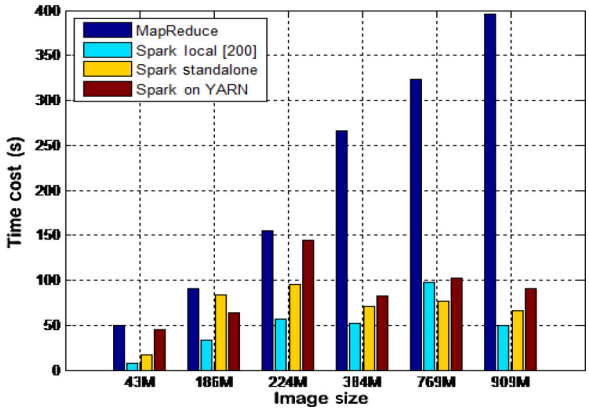

Fig. 8. Efficiency of Sobel operator on different images with 3*3 convolution.

D. Efficiency of Pixel-Level Parallel Algorithms

1) Pixel-Level Parallel: We designed a Spark algorithm that changed pixel digital numbers and count numbers of strips of images. On a Spark standalone model, the algorithm took 1110.47 s to modify each pixel of 6 988 800 strips from 117.1 GB images, while 3590.04 s were taken to modify each pixel of 10 368 000 strips from 405.8 GB images, and 224.27 s were spent to modify each pixel of 3 192 000 strips from 37.4 GB images. On Spark on YARN model, it took 1754.41, 3886.98, and 356.52 s, respectively. It appears that 44 811 756 pixels can be processed per second. Further, 48 518 234 pixels can be processed per second with the Spark standalone model. The results showed that employment of a Spark standalone model can be 1.08–1.6 times faster than use of Spark on a YARN model being used by pixel parallel algorithms. The reason is that the former model has more a fine-grained resource management policy than does the latter. The test results can be seen as the initial cost of strip-oriented multitasking algorithms; we built the performance benchmark for this type of algorithms. The results show that about 45 million pixels can be processed in a second on the small cluster. The strip-oriented parallel processing method is thus efficient and suitable for the type of algorithms shown in Table IV.

2) Neighboring Pixel Processing Algorithms: Image convolution is a type of neighboring pixel processing. For example, there is the widely used Sobel edge detection operator which calculates gradient using a 3*3 window for each pixel of original images. Image smoothing requires relatively large convolution windows to average the pixel value of the original images. Deep convolution neuron networks require basic operations to extract abstraction features from low-level features [62]. Therefore, this type of algorithms is of great importance in RS algorithms. In this test, we compared the efficiency of the two types algorithms with four executing models (MapReduce, Spark Local [200], Spark standalone, and Spark on YARN). The first algorithm is the Sobel operator. The second is the Gaussian image smoothing algorithm (GISA) with different window sizes (5*5, 7*7, and 9*9). The target RS data are of different sizes ranging from 43 to 909 M with resolutions from 5271*4318 to 16 761*14 421. We implemented two versions of the algorithms, using both MapReduce and our strip-programming model. Parallel Sobel operator on images using our model can be implemented using three filtering, two joining, and three mapping transformation primitives. Supposing an image has N rows and M columns, and each row is small enough to fit in memory. The three filtering primitives are used to obtain three RDDs that represent 0 to N-2 rows of the image represented as RDD1, 1 to $N-1$ rows of the image represented as RDD2 and 2 to N rows of the image represented as RDD3, respectively. Before applying two joining transformation primitives, two mapping transformation would be applied on RDD2 and RDD3 to change their row numbers to 0 to $N-2$. Rows that have the same keys are then joined together using two joining primitives so that the final RDD can be used to parallel apply Sobel functions on multiple rows need. Although the implementation is not take consideration of edge effect, the example shows that Strip parallelization can tackle the type of parallel. GISAs were also implemented using the strip parallelization method. MapReduce-based Sobel operator had the worst performance. Even when processing a 43 MB image, it took nearly 50 s. As shown in Table VI, convoluting a 909 MB image took more than 1330 s with 9*9 windows using MapReduce. Convoluting a 909-MB image took 423 s using Spark local [200]. This is mainly because the logical image partition method used in these algorithms introduces an unconstrained network communication cost. As discussed earlier, "WholeFileInputFormat," frequently used in image processing, introduces a zero partition policy [14]. Therefore, only one mapper instance will be generated to first read all image data and to then process each image in single node. Thus, it is much more inefficient as expected. The "Spark local [200]" model here exhibited a compelling advantage over MapReduce. This model is similar to a multiple-threads model, in which the Spark master and Spark executors are all in one machine. Although it is not reasonable to compare MapReduce with this model, the results show obvious performance gains when using Spark rather than MapReduce for rapid image processing. Note that we randomly chose a machine of the cluster environment for the "Spark local [200]" model. As shown in Tables V, VI and Fig. 8, convolution of a 909-MB image with 3*3 window using Spark on YARN model took 90.14 s, while that using MapReduce model took 395.92 s. Convolution of a 909-MB image with 9*9 window using Spark on the YARN model took 353.35 s, while that using MapReduce model took 1332.74 s. MapReduce-based Gaussian smoothing algorithms performed rather poorly, with efficiency degrades dramatically accompanying the growing image size and convolution window size. In contrast, the Spark-based algorithms exhibited strong robustness. The results demonstrate that Spark on a YARN model

TABLE V
TIME COST OF 3*3 SOBEL OPERATOR USING DIFFERENT EXECUTING MODELS

Image size (M)/resolution	Spark local [200] (s)	Spark standalone (s)	Spark on YARN (s)	MapReduce (s)
43.4/(5271*4318)	7.30	17.01	44.98	49.43
186/(16143*12082)	32.78	84.18	63.74	90.12
224/(16691*14114)	56.46	95.67	144.79	155.55
384/(16800*12000)	51.80	70.43	82.60	266.16
769/(16800*12000)	97.61	76.69	102.36	323.84
909/(16761*14221)	50.02	66.60	90.14	395.91

TABLE VI
TIME COST OF 9*9 GAUSSIAN IMAGE SMOOTHING USING DIFFERENT EXECUTING MODELS

Image size (s)/resolution	Spark local [200] (s)	Spark standal one (s)	Spark on YARN (s)	MapReduce (s)
43.4/(5271*4318)	47.82	69.23	109.11	156.80
186/(16143*12082)	223.36	211.73	223.80	293.50
224/(16691*14114)	374.86	486.14	363.83	513.29
384/(16800*12000)	428.44	268.04	285.98	587.31
769/(16800*12000)	798.96	392.40	434.79	926.47
909/(16761*14221)	422.94	252.47	253.35	1332.74

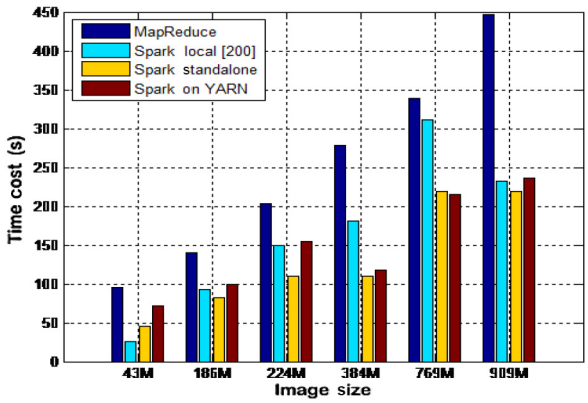

Fig. 9. Efficiency of Gaussian image smoothing with 5*5 windows size.

is about five times faster than MapReduce when being used in neighbor-based parallel algorithms (NBPAs) on the cluster. When image size and convolution window size increase, Spark-based algorithms may need more time to complete. However, there is no significant performance degradation using Spark on YARN model. As shown in Figs. 9–11, GISAs with 5*5 windows cost at most 250 s, GISAs with 7*7 windows cost less than 300 s, while GISAs with 9*9 windows cost less than 440 s. The results show that Spark on YARN model is rather robust for NBPAs. The most efficient model for NBPAs is the Spark standalone model. When using the model, GISAs with 5*5 windows cost less than 250 s, GISAs with 7*7 windows cost less than 300 s, while GISAs with 9*9 windows cost less than 500 s. In fact, Spark on the YARN model requires plenty of time to upload relevant libraries and configuration files to HDFS and should negotiate with ResourceManager to build the initial environment for Spark engine. Furthermore, the Spark standalone model and Spark on YARN adopt different mechanisms to manage resource abstraction. Thus, it may be unjust to conclude that the former is better than the latter for NBPAs. As

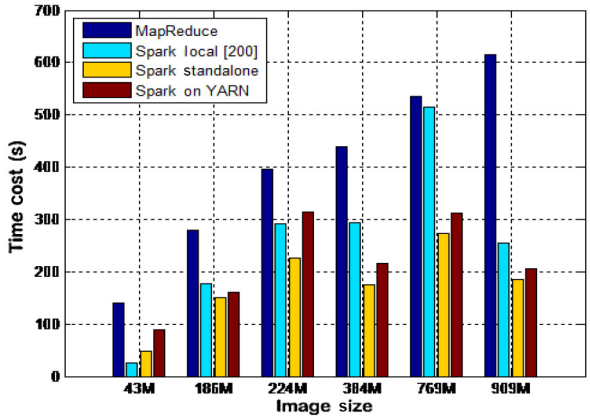

Fig. 10. Efficiency of Gaussian image smoothing with 7*7 windows size.

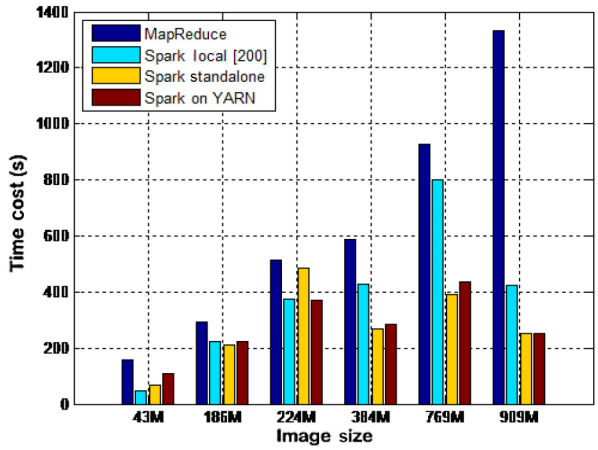

Fig. 11. Efficiency of Gaussian image smoothing with 9*9 windows size.

shown in Table VI, convoluting a 224-MB image took 363.83 s, whereas convoluting a 909-MB image took only 253.35 s. The result shows no linear relationship between image size and time cost of convolution algorithms when using the Spark on YARN model. The larger the image size, the more probable it is that task-scheduling in the Spark engine will dispatch more tasks to the data nodes where the image content resides. All of these results testified that the Spark on YARN model is suitable for NBPAs. The window size is the main factor that affects the efficiency of the type of algorithms.

E. Efficiency of BLPAs

Regarding the BLPAs, the efficiency of NDVI was leveraged to investigate the Spark on YARN model against two datasets. The first dataset consisted of a large number of RS images (536), each of 96.2 MB, and the second consisted of a small number of RS images (42) with each image being 1.2 GB. The images in the first dataset are single band images. The red and near-infrared bands were extracted from multiple-spectral images. Because the two datasets have rather different images, we built two BLPAs called BLPA1 and BLPA2 to calculate NDVI, respectively. The algorithms were executed 10 times, and the average values of the runs were recorded. As expected, the ratio column in Table VII shows that BLPA1

TABLE VII
EFFICIENCY OF THE BAND-LEVEL PARALLEL ON TWO DATASETS WITH SPARK ON YARN

Container number	DataSet1 (536*96.2 M) (s)	DataSet2 (42*1.2 GB) (s)	Ratio
120	5348.42	959.88	5.57
100	3743.43	824.53	4.54
80	4363.77	932.43	4.68
60	4158.58	875.49	4.75
40	3500.21	806.50	4.34
20	3461.55	635.33	5.45

would be four to six times slower than BLPA2 to process the same volume of RS data. BLPA1 used two mapping operations and one joining operation. Shuffle operations taken by Spark engine would be costly. BLPA2 can be regarded as a pixel-level algorithm and only needs one mapping operation. The red and near-infrared bands wrapped in strips of multiple-spectral images can be parallel processed without moving any data blocks. The result indicated that the efficiency of BLPAs is mainly determined by the number of joining operations rather than the volume of the dataset; furthermore, it testified that the strip-oriented parallel process method is feasible for BLPAs to process large volumes of multiple-spectral and hyper-spectral images. Specifying 120 containers for BLPA1, the time cost was 5348.42 s, while when specifying 20 containers for BLPA1, it took 2422.91 s. Specifying 120 containers for BLPA2, the time cost was 959.88 s, while when specifying 20 containers for BLPA1, the time cost was 635.33 s. As container number decreases, the performance of the algorithms inversely increased in the cluster. To conclude this section, container number is not a determined factor to BLPAs. In fact, when more containers are available, the Spark engine would spend more spaces and time to store and maintain the status of Spark executors. Moreover, in a relatively small cluster, containers may be distributed evenly. Thus, increasing the containers for Spark-based algorithms is not useful to greatly enhance the performance of BLPAs.

F. Efficiency of Irregular Parallel Algorithms

The widely used K-means clustering algorithms were executed 10 times using the Spark on YARN model in this test. When the desired cluster number is set, K-means clustering assigns each pixel to one of the clusters according to the principle that the distance between each pixel of a cluster and its centroid must be smaller than the distance from the pixel to any other cluster centroids. Since, the whole image information should be used in the clustering procedure [47], it is both data-intensive and iteration-intensive, and it can represent the irregular-level parallel RS algorithms (ILPAs). Since Spark has built-in implementation of K-means in its machine learning (ML) libraries known as MLlib [63], we can test the efficiency of the type of ILPAs very conveniently. The strip-oriented programming model enables images being mapped to ROWMATRIX [64] and thus Spark-based RS algorithms can seamlessly be integrated with MLlib. The detail of K-means using our programming

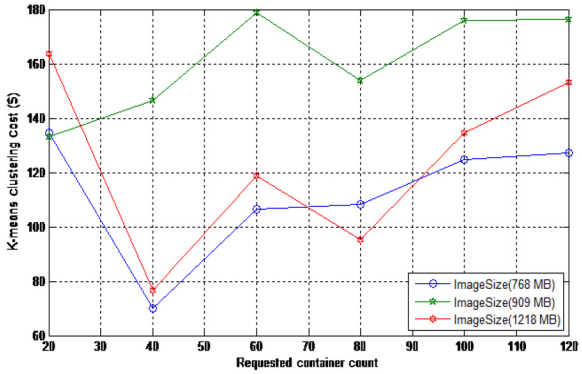

Fig. 12. Impact of the requested container count to K-means clustering cost(s).

model is as following: first, using hadoopFile interface and the aforementioned RsInputFormat class, RS images stored in HDFS can be represented as objects of "JavaPairRDD<Text, BytesWritable>" type, in which Text represents the keys of strips and BytesWritable represents values of corresponding strips. And then, images content can then be mapped to objects of JavaRDD<Vector> type, in which Vector is a data type in Spark Mlib. Finally, models can be obtained by calling relevant interfaces from Mlib library. We can then compare the time costs when assigning them different numbers of YARN containers. The parameters of the K-means algorithms were initially set as $\boldsymbol{K} = 2$, $\boldsymbol{maxiteration} = 20$, which represented, respectively, the number of desired clusters and the maximum number of iterations to run. Three different images (768, 909, and 1218 MB) were employed. As shown in Fig. 12, the K-means algorithms completed in less than 180 s. The general trend of time cost of K-means algorithms becomes flat when the number of containers surpasses 100. Overall, clustering the 909 MB image costs more time than that of clustering the other two images. Clustering the 798 MB image cost less than 140 s. Time costs for clustering 1218 MB image have the same trend as those for clustering 768 MB. When using 40 containers, clustering the 1218 MB image required 76.56 s. When the container number increases to 120, clustering the 1218 M image required over 150 s. The results show that ILPAs obtain good performance, and that image size has limited influence on ILPAs. After increasing the number of containers, ILPAs will be more robust. However, the scheduling cost will also increase as discussed earlier.

In order to examine whether the complexity of the algorithms would impact the performance gain, we kept going on our tests. Since the efficiency of clustering algorithms is highly correlated with clustering initialization parameters, it is reasonable to use parameter $\boldsymbol{K}$ to represent the complexity of the type of algorithms. As shown in Fig. 13, clustering the two images (768 MB and 1.21 GB) into 10 categories completed in less than 350 s. However, clustering the 909 MB image took nearly 500 s. Clustering algorithms that processed the 1218 MB image and the 768 MB image cost less time and were more robust than in processing the 909 MB image. The possible reason is that the 909 MB image was partitioned and stored in less powerful nodes. The Spark engine would dispatch more tasks

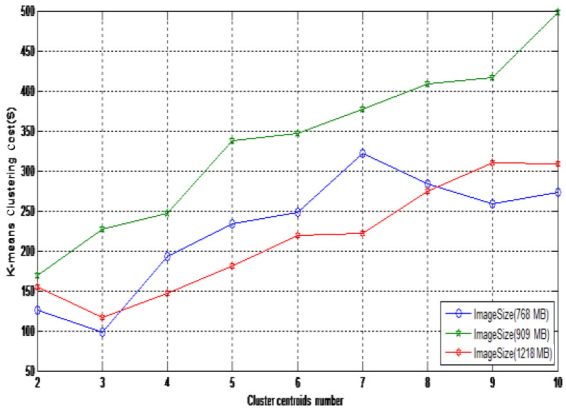

Fig. 13. Impact of cluster centroids number to K-means clustering cost using Spark on YARN model to process large RS data.

to these nodes. Further, the current version of Spark does not have any optimized caching mechanism. Nonetheless, all of the results show that it is quite efficient for integrated processing and analysis of large RS data. ILPAs tend to be quite robust when more containers are available.

V. Discussion

To integrate processing and analysis of large-scale RS data, we employed the in-memory computing framework known as Apache Spark in this study. As is known, GPUs are another widely used hardware for timely processing RS data in many works [65]–[67]. However, programming against the special hardware does require a large number of thread-specific strategies to explicitly transfer data between the PCI devices. Message passing interface (MPI) and CUDA libraries are widely adopted in RS algorithms to promote their efficiency [68]. In addition, applying the programming method is rather difficult and requires well-designed scheduling algorithms to balance workloads between CPU and GPU [65]. Therefore, the programming model is not suitable for integrating processing and analysis of large-scale RS data. Spark-based RS algorithms employ high-level programming model and would be easily implemented as introduced early. Since Spark leverages RDDs to represent data partitioned across distributed nodes, transformation functions are capable of being serialized to each node and executed locally. Therefore, the cost of explicit data transfers of large data blocks between processing units is eliminated. The Spark engine has concealed much scheduling and fault handler detail too. Thus, scientists can concentrate on algorithmic logic rather than low-level communication control technologies and task-scheduling.

Hadoop has become the most widely used platform for distributed storage and parallel processing large volumes of data. One of its successors, Hadoop YARN, can dynamically lease containers to frameworks with high reliability and fault tolerance capacity, and is widely used in cloud computing environments. This powerful platform for data operations was also used in this work. Other well-known GCP also adapt services-oriented architectures for hiding the variety of computing nodes. However, Hadoop YARN leverages virtualization technologies rather than web services to this end. Thus, it may be more efficient than grid computing frameworks. Since the data access strategy is not location-based in grid platforms, large datasets have to be explicitly transferred with the GridFTP protocol if programming services in grids [11]. While YARN containers are an on-demand computing resource and data stored in HDFS that can be parallel processed with MapReduce programming model. Thus, it may be more feasible. However, the efficiency of MapReduce-based RS algorithms is highly correlated with well-designed logical images partition, which tortured both programming problems and I/O issues as discussed early.

A. Concerning of Generic Image Partition

To efficiently process large-scale RS data that stored in HDFS using Spark-based algorithms, one has not only to decide parallel processing units referenced by RDDs objects at Spark runtime, but also to consider the characteristics of underlying platforms. To better classify our strip-oriented image partition scale is superior to other alternatives such as bands-oriented scale, we compare the efficiency of LSA using different image partition scales in this section. As discussed earlier, LSA is a joining-intensive algorithm that includes five joining operations. There were 72 GLASS02A01 images used in the test and each was 96.2 MB. These images are of HDF4 format. To generate differentiable bands/strips from HDFS for Spark-based algorithm, we implemented the "RecordReader" class that reads bands/strips from the images using NetCDF 4.6.4 library. NetCDF library can only operate on local file system. To overcome the limitation, Hadoop interfaces can be used to read these images to memory buffer, and then use NetcdfFile.openInMemory function to obtain a handle for interpreting and reading bands/strips. Spark-based LSA algorithms using band-oriented programming model were rather similar with that using the strip-oriented programming model. Only keys should be carefully used in joining primitives. Each submission of LSA to YARN cluster was executed 10 times and the average execution time was recorded. We used the top five machines listed in Table I in this test. The YARN cluster could provide 246 GB memory at most. We set the driver-memory to 6 g in each submission of the LSA to fully consume all the cluster memory with executors setting as shown in Table VIII.

LSAs using strip as image partition scale showed an advantage over those using bands as the scale. When 240 Spark executors were used and each had 1 GB memory capacity, the band-oriented LSA (BLSA) algorithms failed in all the tests, while the strip-oriented LSA (SLSA) algorithms took an average of 458.97 s. "Executor lost" was the major error type in the BLSAs. When using 6 GB executors, BLSAs took an average of 404.97 s to complete. The results indicated that Spark is more suitable for small in size tasks. BLSAs would parallel transform on bands, which would require more memory space to store intermediate results. Executors often suffered Java heap error. The size of their tasks would be larger than those of SLSAs. When using 40 executors and each had 6 GB memory, SLSAs took an average of 280.82 s to complete. While

TABLE VIII
TIME COST OF LSA ALGORITHMS USING SPARK/STRIPS AND SPARK/BANDS ON YARN PLATFORM

Partition scale	Execution model	Elapsed time (s)	Executors setting (GB)
Strip	Spark on YARN	458.97	240/1
Band	Spark on YARN	Failed	240/1
Strip	Spark on YARN	271.15	120/2
Band	Spark on YARN	Failed	120/2
Strip	Spark on YARN	272.55	60/4
Band	Spark on YARN	Failed	60/4
Strip	Spark on YARN	280.82	40/6
Band	Spark on YARN	404.97	40/6

TABLE IX
TIME COST OF K-MEANS ALGORITHMS USING SPARK/STRIPS AND SPARK/BANDS ON YARN PLATFORM

Partition scale	Execution model	Elapsed time (ms)	Executors setting
Strip	Spark on YARN	79417	240/1
Band	Spark on YARN	Failed	240/1
Strip	Spark on YARN	50671	120/2
Band	Spark on YARN	Failed	120/2
Strip	Spark on YARN	45891	60/4
Band	Spark on YARN	Failed	60/4
Strip	Spark on YARN	36713	40/6
Band	Spark on YARN	Failed	40/6

BLSAs took an average of 404.97 s. The results indicated that Spark/strip is more suitable and efficient than Spark/band in the joining-intensive algorithms. To classify that our Spark/strip model is more suitable for ILPAs than Spark/band is, we continued our tests with the aforementioned K-means algorithms. The target RS image is 928 MB and consists of 16 bands. The 16 band files had been extracted and uploaded to HDFS previously. The parameters of the K-means algorithms were set as $\boldsymbol{K} = 2$ and $\boldsymbol{maxiteration} = 20$. Each submission of K-means was executed 10 times and the average time cost was recorded. We observed that K-means algorithms using the band-oriented model failed in all the tests as shown in Table IX, while whose using the strip-oriented model demonstrated a good efficiency, taking about 36 s to cluster the image into two classes. In fact, Spark adapts row-based distributed matrix computations as discussed before. Our strip-oriented model enables the mapping of images to ROWMATRIX, and therefore high performance could be obtained. We did not conduct our tests that using pixel as the image partition scale. As discussed earlier, generating pixels from HDFS would suffer great latency to Spark-based RS algorithms. Spark streaming enables stream processing of data streams. However, it is not easy to stream pixels from HDFS.

To conclude, we transform the logic partition problem into pixel selection and localization problem. In Spark-based RS algorithms, we only need to generate differential strips from HDFS blocks and then apply transformations on these strips. Strips in RDD partitions can be employed to localize pixels and bands. Furthermore, Strip-oriented parallel algorithms have no concerns of OOM error. Because strips are small units for most of computing nodes. The advantage of our strip-oriented programming model is that all-level parallel algorithms can be expressed. The impressive efficiency of Spark-based algorithms for rapid processing massive RS data has been tested. Nearly half terabytes of RS data transformed by multitasking algorithms completed in less than 4 h on the small YARN cluster. We did not compare the efficiency of these algorithms with MapReduce-based multitasking algorithms and corresponding serial algorithms in this work. As discussed earlier, chaining separate MapReduce algorithms must use HDFS as intermediate data cache. Therefore, heavy workload must be paid to the additional I/O operations. Traditional serial algorithms either suffer the shortage of memory overflow errors or are tapped in dense disk I/O; thus, their efficiency is rather low as compared with parallel algorithms. The tasks failures rates of these Spark-based algorithms were very lower. According to our experiments, there were at most 3% errors in the stages of these algorithms. Package loss in shuffle stages are the main error source. There were no OOM errors in indices computing stages in our experiments. Although there will be 65%–97.5% runtime costs used by underlying shuffle system in Spark-based algorithms according to our experiments, many works have been proposed to enhance the performance of the shuffle system [33]. Furthermore, JVM parameters and Spark system parameters could be better tuned to reduce GC overhead occurs in shuffle stages of these joining-intensive algorithms [69]. Therefore, it can be expected that future version of Spark will be more efficient. Another merit of strip-oriented paradigm is that large data can be shared between Spark executors for multitasking algorithms. One of our experiments showed that it took less than 2 min to collect and share 2.62 GB RS data. Because the values of these broadcast variables distributed in Spark master and executors and can be localized with BitTorrent protocol, thus, Spark executors can fetch pixels rather efficiently. As long as the configured memory size of the Spark master instance is large enough and the YARN platform can fulfill the request, quite large data can be shared. Using the strip-programming model, RS images stored in HDFS can be mapped to distributed matrices. Spark has inherent computation and optimization of distributed matrix [64]. In-memory parallel processing and analysis of RS big data are of integration. Furthermore, many corporation has open-sourced their Spark-based ML algorithms [70]. FVC, LST, and LSA indices parallel extracted from huge volume of RS data and *in situ* soil measurements can then be used to train useful models for projecting surface soil moistures.

B. Comparing With Spark on Apache Mesos Model

Another widely used platform that can also efficiently provide highly available and fault-tolerant cluster resources to Spark—the Apache Mesos platform [71]—was selected for comparison with Spark on the YARN model. Being different to the YARN platform that is dedicated to providing multiple

TABLE X
TIME COST OF LSA ALGORITHMS USING SPARK ON YARN AND SPARK ON MESOS

Execution model	Elapsed time (s)	Executors setting
Spark on YARN	4645.97	--num-executors 240 --executor-memory 1 g
Spark on YARN	4237.15	--num-executors 120 --executor-memory 2 g
Spark on YARN	3210.41	--num-executors 60 --executor-memory 4 g
Spark on YARN	2536.19	--num-executors 40 --executor-memory 6 g
Spark on Mesos	4524.49	--executor-memory 30 g --total-executor-cores 240
Spark on Mesos	4437.12	--executor-memory 40 g --total-executor-cores 240

TABLE XI
TIME COST OF K-MEANS ALGORITHMS USING SPARK ON YARN AND SPARK ON MESOS

Execution model	Elapsed time (s)	Executors setting
Spark on YARN	79.66	--num-executors 240 --executor-memory 1 g
Spark on YARN	53.04	--num-executors 120 --executor-memory 2 g
Spark on YARN	51.14	--num-executors 60 --executor-memory 4 g
Spark on YARN	42.27	--num-executors 40 --executor-memory 6 g
Spark on Mesos	83.82	--executor-memory --total-executor-cores 240
Spark on Mesos	72.26	--executor-memory 2 g --total-executor-cores 240
Spark on Mesos	87.58	--executor-memory 3 g --total-executor-cores 240
Spark on Mesos	90.60	--executor-memory 4 g --total-executor-cores 240
Spark on Mesos	116.80	--executor-memory 5 g --total-executor-cores 240
Spark on Mesos	174.67	--executor-memory 10 g --total-executor-cores 240
Spark on Mesos	154.40	--executor-memory 20 g --total-executor-cores 240
Spark on Mesos	Failed	--executor-memory 30 g --total-executor-cores 240

frameworks with on-demanding containers for big data processing applications, Mesos aims to provide a core for enabling multiple frameworks such as Spark, MPI, and Hadoop to efficiently share cluster resources. It has the potential to provide elastic resources for distributed computing frameworks. The main component of Mesos consists of a master process and slave daemons running on each cluster node. Having been offered resources after a framework registered with the master process, a *scheduler* of the framework would select satisfactory resource offers. Executors of the framework launched on slave nodes are responsible for executing the framework's tasks. Mesos can scale up to 10 000 nodes and is commonly used in large data center for enabling fault-tolerant and elastic distributed systems to easily be built and run effectively.

The aforementioned five-node YARN cluster was also used in this test. The Mesos master process was configured to run on the first node as was the Hadoop master. Mesos-0.26.0 was used to construct the Mesos cluster without using Zookeeper. Note that Zookeeper [72] is typically used by Mesos for maintaining fault tolerance of the master process. Since the cluster is too small, we did not configure Zookeeper for the Mesos cluster. SLSAs were used to process about 33.8 GB GLASS02A01 images in the tests. Each submission of the algorithms was executed 10 times and the shortest runs were recorded. Being different from Spark on the YARN model that can specify both the executor number and per-executor-memory size, Spark on the Mesos model can only coarsely specify the total executor's cores and per-executor-memory size using "–total-executor-cores" and "–executor-memory" parameters concurrently. We attempted to make the two models comparable. As shown in Table X, the shortest run of SLSAs using Spark on the Mesos model took 4437.12 s, while the shortest run using Spark on the YARN model completed in 2536.19 s. The results indicated that Spark on the YARN model can be 1.7 times faster than Spark on Mesos when processing joining-intensive Spark-based RS algorithms. As discussed earlier, Spark on the YARN model preferably use memory to store RDDs partitions for following transformations. The shortest run can be obtained when each executor is given 6 G memory. However, when using a 1-GB per-executor setting with a large number of executors, about 1.2 h would be used to complete. Although the increasing number of executors adds additional scheduling overhead to the Spark engine, we observed that the efficiency of the algorithms using Spark on the YARN model was comparable with those using Spark on the Mesos model. We also tracked the shortest runs of the algorithms and found that over 75% of the tasks completed in less than 16 s when using Spark on the YARN model, while only 45% could be completed in less than 16 s when using Spark on the Mesos model. We also tested the efficiency of the aforementioned K-means algorithms using the two execution models. Each submission of the algorithm was executed five times and the average executing time was recorded. As shown in Table XI, the average time cost of the algorithm was 72.26 s when using Spark on the Mesos model, while it was 42.27 s when using Spark on the YARN model. The results indicated that iterative-intensive algorithms using Spark on YARN can be 1.7 times faster than those using Spark on the Mesos model. Moreover, increasing memory size for the algorithms using Spark on Mesos does not increase their performance. When "–executor-memory" was set to 1 GB, it took about 83.82 s to complete. But when the parameter was set to 20 GB, it would take about 154.4 s. This is largely because that the Spark runtime would get less chance to accept usable resources from Mesos when the executor-memory size was set too large. When the parameter was increased to 30 GB, K-means algorithms failed all the time. Form the Mesos log file, we found that tasks of the algorithms that failed due to Mesos cannot allocate memory to containers for the Spark engine. In fact, YARN and Mesos are two similar frameworks that support sharing resources among different frameworks. The difference between the two platforms lies in their resource scheduling mechanism. After being requested resources from frameworks, YARN would evaluate all available resources and place jobs, while Mesos pushing controls to frameworks decides whether to accept or reject the resource offers. We do not plan to

claim which platform is better. YARN as the next generation of MapReduce is more suitable for both joining-intensive and iterative-intensive Spark-based RS algorithms as testified by our experiments. We still do not know the memory isolation and utilization mechanism when using Spark on Mesos. It seems that Mesos would first judge whether a host has sufficient memory that is bigger than the requested executor-memory size, and then Spark executors would be only launched on the slaves. When we set "executor-memory" parameter to 1 GB in the SLSA submission using Mesos, only 6.9 GB cluster memory spaces were used. When we set the parameter to 30 GB, only 132.0 GB were used, and setting to 40 GB would use only 176.0 GB. Although there were a total of 256.3 GB memory, we cannot easily set a policy to utilize all available resource when using Spark on the Mesos model for Spark-based algorithms.

Many may argue that a strip-oriented programming model may not be suitable for joining-intensive algorithms because shuffling is the most time-consuming operation of the system. The shuffle writing operation that occurs before the joining stages need to write a large volume of data, and the shuffle reading operation needs to read a huge volume of data in the joining either. However, by tracking shuffling stages of the LSA algorithms that process 33.8 GB RS images in the five nodes cluster, we observed that there were 1282 MB to 3.7 GB of data written to disk within 15 min before the QC stages, while in the QC stages, there were only 4.9–10 GB data to read by Spark executors within 3 min. Also, no tasks failure occurs in the shuffling stages. Typically, the shuffle writing data volume was quite small and the time cost of the shuffle reading stages was quite low. This is largely because of the shuffle consolidate policy used by Spark and Netty supports very efficient networking for Spark to pull the data from disks as previously mentioned. Since YARN provides on-demand containers for Spark, we can expect that the time cost is affordable for algorithms that process large amounts of RS data in the cloud.

VI. Conclusion and Future Work

In this paper, we propose a novel parallel programming model for integrated processing and analysis of large-scale RS data using Spark on a Hadoop YARN platform. By incorporating strip abstraction of RS data and Spark RDDs, all-level RS parallel algorithms can be expressed irrespectively of the variety of image formats and efficiently executed in a heterogeneous cloud computing environment. RS algorithms can be parallelized with mapping, joining, and filtering primitives and P2P-based broadcast variables. Our experiments indicate that Spark-based RS algorithms can efficiently process large-scale volume of RS data even when data volume exceeds memory capacity. It was possible to process 45 million pixels in 1 s on a small heterogeneous cluster (nine nodes). Applying complex transformations on more than half a terabyte of RS data completed in a few hours. Sharing a large volume of data in multitasking RS algorithms has limited impact on the efficiency of Spark-based RS algorithms. Nearly 51.3% memory would be employed to cache data and execute tasks, and no more than 3% errors would occur in data-intensive computation stages. Spark-based RS algorithms can perform efficiently, and image sizes have limited influence on them. Data-intensive irregular parallel algorithms can be more robust and efficient if given more YARN containers. Compared with Spark on Mesos model, Spark on YARN model can be 1.7× faster for both joining-intensive and iterative-intensive RS algorithms. The strip-oriented programming enables RS images being mapped to distributed vectors and, thus, parallel RS algorithms based on the model can seamlessly integrate with Spark MLlib. Moreover, they can be easily transplant to other YARN clusters without any modification for intercluster processing RS big data. Our future work will focus on optimizing a global cache for Apache Spark and on deep learning from massive RS data for estimation of global-scale soil moisture for the RS community.

References

[1] J. Famiglietti, A. Cazenave, A. Eicker, J. Reager, M. Rodell, and I. Velicogna, "Satellites provide the big picture," *Science*, vol. 349, pp. 684–685, 2015.

[2] Y. Ma *et al.*, "Remote sensing big data computing: Challenges and opportunities," *Future Gener. Comput. Syst.*, vol. 51, pp. 47–60, 2015.

[3] H. Xia, H. A. Karimi, and L. Meng, "Parallel implementation of Kaufman's initialization for clustering large remote sensing images on clouds," *Comput. Environ. Urban Syst.*, 2014, in progress, Corrected Proof.

[4] J. Li, L. Meng, F. Z. Wang, W. Zhang, and Y. Cai, "A map-reduce-enabled SOLAP cube for large-scale remotely sensed data aggregation," *Comput. Geosci.*, vol. 70, pp. 110–119, 2014.

[5] L. Mascolo, M. Quartulli, P. Guccione, G. Nico, and I. G. Olaizola, "Distributed mining of large scale remote sensing image archives on public computing infrastructures," arXiv preprint arXiv:1501.05286, 2015.

[6] R. Giachetta, "A framework for processing large scale geospatial and remote sensing data in map reduce environment," *Comput. Graph.*, vol. 49, pp. 37–46, 2015.

[7] Z. Sun, F. Chen, M. Chi, and Y. Zhu, "A spark-based big data platform for massive remote sensing data processing," in *Data Science*, New York, NY, USA: Springer, 2015, pp. 120–126.

[8] R. Giachetta and I. Fekete, "A case study of advancing remote sensing image analysis," *Acta Cybernetica*, vol. 22, pp. 57–79, 2015.

[9] C. Lee, S. D. Gasster, A. Plaza, C.-I. Chang, and B. Huang, "Recent developments in high performance computing for remote sensing: A review," *IEEE J. Sel. Topics Appl. Earth Observ. Remote Sens.*, vol. 4, no. 3, pp. 508–527, Sep. 2011.

[10] F. Van Den Bergh, K. J. Wessels, S. Miteff, T. L. Van Zyl, A. D. Gazendam, and A. K. Bachoo, "HiTempo: A platform for time-series analysis of remote-sensing satellite data in a high-performance computing environment," *Int. J. Remote Sens.*, vol. 33, pp. 4720–4740, 2012.

[11] G. Giuliani, N. Ray, and A. Lehmann, "Grid-enabled spatial data infrastructure for environmental sciences: Challenges and opportunities," *Future Gener. Comput. Syst.*, vol. 27, pp. 292–303, 2011.

[12] M. Peter and G. Timothy, "The NIST definition of cloud computing," NIST special publication, 800-145, U.S. Department of Commerce, vol. 9, 2011.

[13] P. Wang, J. Wang, Y. Chen, and G. Ni, "Rapid processing of remote sensing images based on cloud computing," *Future Gener. Comput. Syst.*, vol. 29, pp. 1963–1968, 2013.

[14] M. H. Almeer, "Cloud hadoop MapReduce for remote sensing image analysis," *J. Emerg. Trends Comput. Inf. Sci.*, vol. 3, pp. 637–644, 2012.

[15] F.-C. Lin, L.-K. Chung, C.-J. Wang, W.-Y. Ku, and T.-Y. Chou, "Storage and processing of massive remote sensing images using a novel cloud computing platform," *GISci. Remote Sens.*, vol. 50, pp. 322–336, 2013.

[16] P. Bajcsy, A. Vandecreme, J. Amelot, P. Nguyen, J. Chalfoun, and M. Brady, "Terabyte-sized image computations on hadoop cluster platforms," in *Proc. IEEE Int. Conf. Big Data*, 2013, pp. 729–737.

[17] Y. Bu, B. Howe, M. Balazinska, and M. D. Ernst, "HaLoop: Efficient iterative data processing on large clusters," *Proc. VLDB Endowment*, vol. 3, pp. 285–296, 2010.

[18] Y. Zhang, Q. Gao, L. Gao, and C. Wang, "iMapReduce: A distributed computing framework for iterative computation," *J. Grid Comput.*, vol. 10, pp. 47–68, 2012.

[19] F. Pan, Y. Yue, J. Xiong, and D. Hao, "I/O characterization of big data workloads in data centers," in *Big Data Benchmarks, Performance Optimization, and Emerging Hardware*, New York, NY, USA: Springer, 2014, pp. 85–97.

[20] K. Kambatla and Y. Chen, "The truth about MapReduce performance on SSDs," in *Proc. USENIX 28th Large Install. Syst. Admin. Conf. (LISA'14)*, 2014, pp. 1–9.

[21] M. Sevilla, I. Nassi, K. Ioannidou, S. Brandt, and C. Maltzahn, "SupMR: Circumventing disk and memory bandwidth bottlenecks for scale-up MapReduce," in *Proc. IEEE Int. Parallel Distrib. Process. Symp. Workshops (IPDPSW'14)*, 2014, pp. 1505–1514.

[22] C. P. Chen and C.-Y. Zhang, "Data-intensive applications, challenges, techniques and technologies: A survey on big data," *Inf. Sci.*, vol. 275, pp. 314–347, 2014.

[23] M. Zaharia, M. Chowdhury, M. J. Franklin, S. Shenker, and I. Stoica, "Spark: Cluster computing with working sets," in *Proc. 2nd USENIX Conf. Hot Topics Cloud Comput.*, 2010, p. 10.

[24] R. Xin, P. Deyhim, A. Ghodsi, X. Meng, and M. Zaharia, "GraySort on apache spark by databricks," Databricks, Sort in Spark, Nov. 2014.

[25] V. K. Vavilapalli *et al.*, "Apache hadoop YARN: Yet another resource negotiator," in *Proc. 4th Annu. Symp. Cloud Comput.*, 2013, p. 5.

[26] M. Zaharia *et al.*, "Resilient distributed datasets: A fault-tolerant abstraction for in-memory cluster computing," in *Proc. 9th USENIX Conf. Netw. Syst. Des. Implement.*, 2012, pp. 2–2.

[27] S. Ghemawat, H. Gobioff, and S.-T. Leung, "The Google file system," in *ACM SIGOPS Operating System Review*. Bolton Landing, New York, USA, 2003, pp. 29–43.

[28] J. Dean and S. Ghemawat, "MapReduce: A flexible data processing tool," *Commun. ACM*, vol. 53, pp. 72–77, 2010.

[29] A. Toshniwal *et al.*, "Storm@twitter," *presented at the Proceedings of the 2014 ACM SIGMOD International Conference on Management of Data*, Snowbird, UT, USA, 2014.

[30] B. Saha, H. Shah, S. Seth, G. Vijayaraghavan, A. Murthy, and C. Curino, "Apache Tez: A unifying framework for modeling and building data processing applications," in *Proc. ACM SIGMOD Int. Conf. Manage. Data*, 2015, pp. 1357–1369.

[31] K. Wang *et al.*, "Overcoming hadoop scaling limitations through distributed task execution," in *Proc. IEEE Int. Conf. Cluster Comput. (CLUSTER'15)*, 2015, pp. 236–245.

[32] H. Karau, A. Konwinski, P. Wendell, and M. Zaharia, *Learning Spark: Lightning-Fast Big Data Analysis*. Sebastopol, CA, USA: O'Reilly Media, Inc, 2015.

[33] A. Davidson and A. Or, "Optimizing shuffle performance in spark," Univ. California, Berkeley-Dept. Electr. Eng. Comput. Sci., UC Berkeley, Tech. Rep, 2013, p. 10.

[34] M. Armbrust *et al.*, "Scaling spark in the real world: Performance and usability," *Proc. VLDB Endowment*, vol. 8, pp. 1840–1843, 2015.

[35] N. Rana and S. Deshmukh, "Performance improvement in apache spark through shuffling," *Int. J. Sci. Eng. Technol. Res. (IJSETR'15)*, vol. 4, p. 3, 2015.

[36] T. White, *Hadoop: The Definitive Guide*. Sebastopol, CA, USA: O'Reilly Media, Inc, 2012.

[37] W. Dai and M. Bassiouni, "An improved task assignment scheme for hadoop running in the clouds," *J. Cloud Comput.*, vol. 2, pp. 1–16, 2013.

[38] J. Xing and R. Sieber, "Sampling based image splitting in large scale distributed computing of earth observation data," in *Proc. IEEE Int. Geosci. Remote Sens. Symp. (IGARSS'14)*, 2014, pp. 1409–1412.

[39] A. Aji *et al.*, "Hadoop GIS: A high performance spatial data warehousing system over MapReduce," *Proc. VLDB Endowment*, vol. 6, pp. 1009–1020, 2013.

[40] N. Ritter *et al.*, "GeoTIFF format specification GeoTIFF revision 1.0," 2000, pp. 1–95 [Online]. Available: http://www.remotesensing.org/geotiff/spec/geotiffhome.html.

[41] M. Folk, A. Cheng, and K. Yates, "HDF-5: A file format and I/O library for high performance computing applications," in *Proc. Supercomput.*, 1999, pp. 5–33.

[42] R. Rew and G. Davis, "NetCDF: An interface for scientific data access," *IEEE Comput. Graph. Appl.*, vol. 10, no. 4, pp. 76–82, Jul. 1990.

[43] Y. Ma, L. Wang, D. Liu, P. Liu, J. Wang, and J. Tao, "Generic parallel programming for massive remote sensing data processing," in *Proc. IEEE Int. Conf. Cluster Comput. (CLUSTER'12)*, 2012, pp. 420–428.

[44] T. N. Carlson and D. A. Ripley, "On the relation between NDVI, fractional vegetation cover, and leaf area index," *Remote Sens. Environ.*, vol. 62, pp. 241–252, 1997.

[45] B.-C. Gao, "NDWI—A normalized difference water index for remote sensing of vegetation liquid water from space," *Remote Sens. Environ.*, vol. 58, pp. 257–266, 1996.

[46] G. H. Ball and D. J. Hall, "ISODATA, a novel method of data analysis and pattern classification," DTIC Document, 1965.

[47] J. A. Hartigan and M. A. Wong, "Algorithm AS 136: A k-means clustering algorithm," *Appl. Statist.*, vol. 28, pp. 100–108, 1979.

[48] F. D. Sacerdoti, M. J. Katz, M. L. Massie, and D. E. Culler, "Wide area cluster monitoring with ganglia," in *Proc. IEEE Int. Conf. Cluster Comput.*, 2003, pp. 289–298.

[49] Cloudear, *Tuning the Cluster for MapReduce v2 (YARN)*, 2014 [Online]. Available: http://www.cloudera.com/documentation/enterprise/latest/topics/cdh_ig_yarn_tuning.html.

[50] Z. Jiang *et al.*, "Analysis of NDVI and scaled difference vegetation index retrievals of vegetation fraction," *Remote Sens. Environ.*, vol. 101, pp. 366–378, 2006.

[51] S.-B. Duan, Z.-L. Li, B.-H. Tang, H. Wu, and R. Tang, "Generation of a time-consistent land surface temperature product from MODIS data," *Remote Sens. Environ.*, vol. 140, pp. 339–349, 2014.

[52] J. Wickham, C. Barnes, M. Nash, and T. Wade, "Combining NLCD and MODIS to create a land cover-albedo database for the continental united states," *Remote Sens. Environ.*, vol. 170, pp. 143–152, 2015.

[53] C. A. Clark and P. W. Arritt, "Numerical simulations of the effect of soil moisture and vegetation cover on the development of deep convection," *J. Appl. Meteorol.*, vol. 34, pp. 2029–2045, 1995.

[54] Y. Ding, X. Zheng, T. Jiang, and K. Zhao, "Comparison and validation of long time serial global GEOV1 and regional Australian MODIS fractional vegetation cover products over the Australian continent," *Remote Sens.*, vol. 7, pp. 5718–5733, 2015.

[55] J. Wang, Y. Zhao, C. Li, L. Yu, D. Liu, and P. Gong, "Mapping global land cover in 2001 and 2010 with spatial–temporal consistency at 250 m resolution," *ISPRS J. Photogramm. Remote Sens.*, vol. 103, pp. 38–47, 2015.

[56] E. Vermote, S. Kotchenova, and J. Ray, "MODIS surface reflectance user's guide version 1.3," in *MODIS Land Surface Reflectance Science Computing Facility*, 2011 [Online]. Available: http://www.modissr.ltdri.org/.

[57] Z. Wan, "MODIS land surface temperature products users' guide," Inst. Comput. Earth Syst. Sci., Univ. California, Santa Barbara, CA, USA, 2006 [Online]. Available: http://www.icess.ucsb.edu/modis/LstUsrGuide/usrguide. html.

[58] Y. Qu, Q. Liu, S. Liang, L. Wang, N. Liu, and S. Liu, "Direct-estimation algorithm for mapping daily land-surface broadband albedo from MODIS data," *IEEE Trans. Geosci. Remote Sens.*, vol. 52, no. 2, pp. 907–919, Feb. 2014.

[59] S. Liang *et al.*, "A long-term Global LAnd Surface Satellite (GLASS) data-set for environmental studies," *Int. J. Digital Earth*, vol. 6, pp. 5–33, 2013.

[60] A. Strahler *et al.*, "MODIS land cover product: Algorithm theoretical basis document," University College of London, London, U.K., 1994.

[61] E. Bartholomé and A. Belward, "GLC2000: A new approach to global land cover mapping from earth observation data," *Int. J. Remote Sens.*, vol. 26, pp. 1959–1977, 2005.

[62] A. Krizhevsky, I. Sutskever, and G. E. Hinton, "ImageNet classification with deep convolutional neural networks," in *Advances in Neural Information Processing Systems*, Neural Information Processing System conference, 2012, pp. 1097–1105.

[63] X. Meng *et al.*, "MLlib: Machine learning in apache spark," arXiv preprint arXiv:1505.06807, 2015.

[64] R. B. Zadeh *et al.*, "linalg: Matrix computations in apache spark," arXiv preprint arXiv:1509.02256, p. 13, 2015.

[65] L. Fang, M. Wang, D. Li, and J. Pan, "CPU/GPU near real-time preprocessing for ZY-3 satellite images: Relative radiometric correction, MTF compensation, and geocorrection," *ISPRS J. Photogramm. Remote Sens.*, vol. 87, pp. 229–240, 2014.

[66] A. Plaza, Q. Du, Y.-L. Chang, and R. L. King, "High performance computing for hyperspectral remote sensing," *IEEE J. Sel. Topics Appl. Earth Observ. Remote Sens.*, vol. 4, no. 3, pp. 528–544, Sep. 2011.

[67] E. Christophe, J. Michel, and J. Inglada, "Remote sensing processing: From multicore to GPU," *IEEE J. Sel. Topics Appl. Earth Observ. Remote Sens.*, vol. 4, no. 3, pp. 643–652, Sep. 2011.

[68] M. Abouali, J. Timmermans, J. E. Castillo, and B. Z. Su, "A high performance GPU implementation of surface energy balance system (SEBS) based on CUDA-C," *Environ. Modell. Softw.*, vol. 41, pp. 134–138, 2013.

[69] T. Chiba and T. Onodera, "Workload characterization and optimization of TPC-H queries on apache spark," IBM Research Report, Tokyo, RT0968, Oct. 16, 2015, p. 12.

[70] M. Crawford, T. M. Khoshgoftaar, J. D. Prusa, A. N. Richter, and H. Al Najada, "Survey of review spam detection using machine learning techniques," *J. Big Data*, vol. 2, pp. 1–24, 2015.

[71] B. Hindman *et al.*, "Mesos: A platform for fine-grained resource sharing in the data center," in *Proc. 8th USENIX Conf. Netw. Syst. Des. Implement. (NSDI'11)*, 2011, pp. 22–22.

[72] P. Hunt, M. Konar, F. P. Junqueira, and B. Reed, "ZooKeeper: Wait-free coordination for internet-scale systems," in *USENIX Annu. Tech. Conf.*, 2010, p. 9.

Wei Huang is currently pursuing the Ph.D. degree in remote sensing and information engineering at Wuhan University, Wuhan, China.

His research interests include cloud computing, deep learning, and their applications on remote sensing.

Lingkui Meng was a Visiting Professor at the joint Centre of Cambridge-Cranfield for High Performance Computing, Cranfield University, Cranfield, U.K., between 2006 and 2007. He is currently a Professor with the School of Remote Sensing and Information Engineering, Wuhan University, Wuhan, China. His research interests include remote-sensing application in hydrology, cloud computing, and big data analysis.

Dongying Zhang is currently pursuing the Ph.D. degree in remote sensing and information engineering at Wuhan University, Wuhan, China.

He was a Visiting Ph.D. student at the Global Environment and Natural Resources Institute, George Mason University, Fairfax, VA, USA, between 2014 and 2015. His research interests include deep learning, remote sensing, and their application in hydrology.

Wen Zhang received the Ph.D. degree from Wuhan University, Wuhan, China, in 2009.

She was a Visiting Scholar at the joint Centre of Cambridge-Cranfield for High Performance Computing, Cranfield University, Cranfield, U.K., between 2007 and 2008. She is currently a Lecture with the School of Remote Sensing and Information Engineering, Wuhan University. She has authored more than 10 peer-reviewed papers. Her research interests include network GIS, remote-sensing applications, and spatial data analysis.