Cluster Computing (2020) 23:397–408
https://doi.org/10.1007/s10586-019-02929-x

Canny edge detection and Hough transform for high resolution video streams using Hadoop and Spark

Bilal Iqbal[1] · Waheed Iqbal[1] · Nazar Khan[1] · Arif Mahmood[2] · Abdelkarim Erradi[3]

Received: 26 May 2018 / Revised: 31 December 2018 / Accepted: 29 March 2019 / Published online: 4 April 2019
© Springer Science+Business Media, LLC, part of Springer Nature 2019

Abstract
Nowadays, video cameras are increasingly used for surveillance, monitoring, and activity recording. These cameras generate high resolution image and video data at large scale. Processing such large scale video streams to extract useful information with time constraints is challenging. Traditional methods do not offer scalability to process large scale data. In this paper, we propose and evaluate cloud services for high resolution video streams in order to perform line detection using Canny edge detection followed by Hough transform. These algorithms are often used as preprocessing steps for various high level tasks including object, anomaly, and activity recognition. We implement and evaluate both Canny edge detector and Hough transform algorithms in Hadoop and Spark. Our experimental evaluation using Spark shows an excellent scalability and performance compared to Hadoop and standalone implementations for both Canny edge detection and Hough transform. We obtained a speedup of 10.8× and 9.3× for Canny edge detection and Hough transform respectively using Spark. These results demonstrate the effectiveness of parallel implementation of computer vision algorithms to achieve good scalability for real-world applications.

Keywords Hough transform · Canny edge detection · Video processing · Spark · Hadoop · MapReduce

1 Introduction

Recent advancements in image and video capturing technologies is significantly contributing to the enormous growth of visual digital data. Most of this data is generated through social media networks, personal and public video cameras, smart-phones, surveillance systems, and various types of smart sensors. This visual data can be used in various automated processes by employing computer vision and image processing algorithms. However, these algorithms pose very serious scalability challenges while processing large amount of visual data. It is mainly because these algorithms are tested on small scale datasets and become complex and require unfeasible execution times as the scale of data increases. To obtain good scalability, often approximate methods are employed which may result in accuracy degradation.

Computer vision deals with automatically identifying high-level understanding from visual data attempting to replicate biological visual systems. It helps to automate various tasks including surveillance, anomaly detection, human activity recognition, and traffic automation. However, processing large scale and massively parallel video streams using traditional computer vision methods is extremely challenging. Moreover, for complex, high dimensional, and large datasets the performance of traditional computer vision methods decline noticeably.

✉ Waheed Iqbal
waheed.iqbal@pucit.edu.pk

Bilal Iqbal
MSCSF15M016@pucit.edu.pk

Nazar Khan
nazarkhan@pucit.edu.pk

Arif Mahmood
arif.mahmood@itu.edu.pk

Abdelkarim Erradi
erradi@qu.edu.qa

[1] Punjab University College of Information Technology, Lahore, Pakistan

[2] Information Technology University, Lahore, Pakistan

[3] Department of Computer Science, Qatar University, Doha, Qatar

Human visual perception depends upon the detection of lines and edges. Not surprisingly, computer vision also exploits edges and lines as fundamental building blocks towards a high-level understanding of visual data. The most notable algorithm for finding edges is the Canny edge detector while the Hough transform is mostly used for line or shape detection. To study the effect on processing time for Canny edge detection and Hough transform algorithms using a typical implementation on a large number of images, we performed experiments on varying number of high resolution images. Figure 1 shows the execution time on increasing number of images for both algorithms. We observe an exponential increase in processing time as we increase the number of images to process the standalone implementations of Canny edge detection and Hough transform.

MapReduce [7] is a well-known programming paradigm for running batch processing based applications in parallel and distributed fashion. A typical MapReduce job divides the input into multiple chunks and then processes them in concurrent map functions and the output of all map functions is sorted and then passed to reducer functions which also run concurrently. Finally, the output of the job is stored in a distributed file system. Apache Hadoop is one of the more widely used implementations of MapReduce. However, Apache Spark [22] is gaining traction mainly because it offers both batch processing and stream processing with better performance. The real-time and in-memory processing capabilities are attractive with many applications which require to process large-scale data in real-time.

Figure 2 shows our proposed system to process large-scale video streams to generate alert and notifications. The edge servers hosted near the video cameras detect the key-frames and store them on the hadoop distributed file system (HDFS), a fault tolerant and scalable file system, where each image contains camera id and time-stamp embedded as a metadata. Then our proposed MapReduce-based implementation of canny edge detection using Spark runs after a specific time interval to process the images and produce edge pixels for each image. Similarly, our Hough transform service independently runs periodically and obtains edge pixel files to perform Hough transform for line detection. Then this information is again stored in HDFS for future processing. A final component performs the application specific high-level computer vision task (object detection, activity identification, etc.) to generate alerts and notifications automatically. In this paper, we implemented Canny edge detection and Hough transform services on cloud and we profiled the performance of these services using a wide range of images.

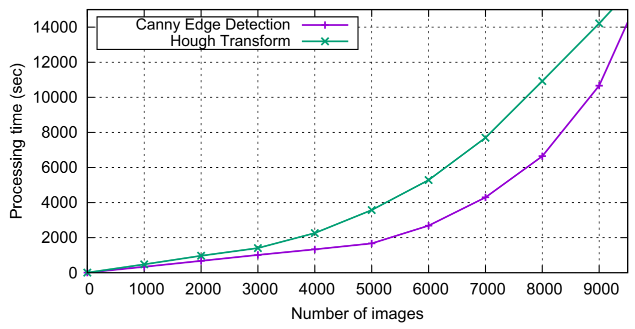


Fig. 1 Canny edge detection and Hough transform processing time increases exponentially as the number of images to be processed increases

The contributions of this paper include:

- A pipeline to process massively parallel video streams is proposed.
- MapReduce/Hadoop and Spark implementations and evaluations of Canny edge detection are performed.
- MapReduce/Hadoop and Spark implementation and evaluations of Hough transform based line detection are performed.
- We evaluated and compared the scalability of Canny edge detection using Hadoop and Spark with a standalone baseline implementation.
- We evaluated the scalability of Hough transform using Hadoop and Spark with a standalone implementation.
- We compared the concurrent versus sequential job executions on Spark cluster for Canny edge detection and Hough transform.

The rest of the paper is organized as follows. Section 2 provides an introduction to MapReduce, Hadoop, and Spark. We discuss related work in Sect. 3. Implementation details for Hadoop and Spark for Canny edge detection and Hough transform are presented in Sect. 4. Experimental evaluation details are given in Sect. 5. Experimental results are presented in Sect. 6. Finally, conclusion and future work are discussed in Section 7.

2 Introduction to MapReduce and Spark

2.1 MapReduce and Hadoop

MapReduce is a programming paradigm introduced by Google for parallel batch processing of large datasets. A typical MapReduce program runs on a cluster of multiple computing nodes with a distributed file system which can store large datasets for processing. A MapReduce program consists of three main phases, Map, Shuffle and Reduce [7].

Map phase consists of a function named map. Each worker node of the MapReduce cluster applies the map function to the part of distributed data stored locally on the

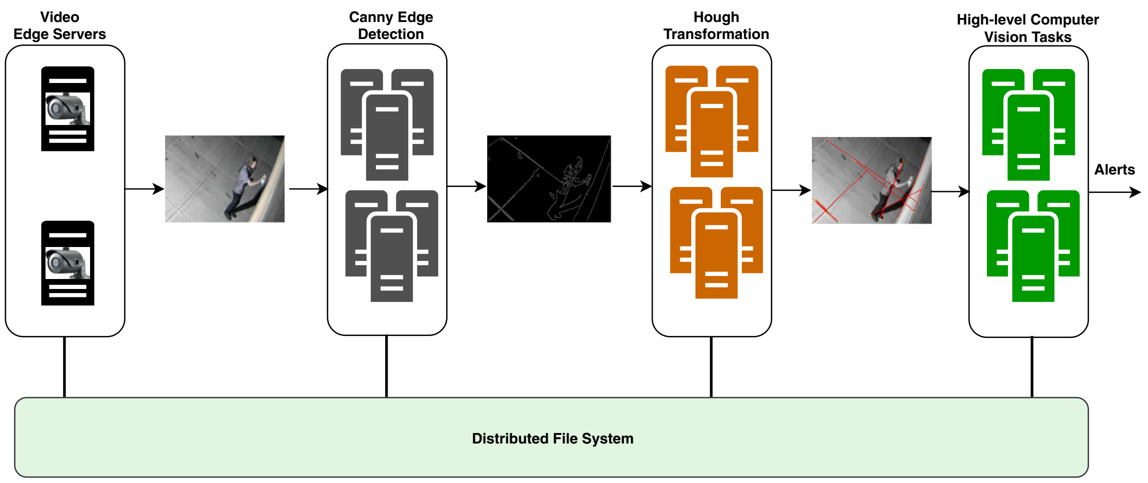


Fig. 2 The proposed scalable pipeline to process large-scale video streams and to generate alerts and notifications. The edge servers hosted closed to the video cameras detect the key-frames and store them to the hadoop distributed file system (HDFS) and then Canny edge detection, Hough transform, and high-level computer vision tasks are executed on the images

node. A map function accepts input as a single key-value pair and outputs a list containing key-value pairs. For a given input key-value pair (k, v), map function will return a set of key-value pairs $\{(k_0, v_0), (k_1, v_1), \ldots, (k_n, v_n)\}$. The map function runs in parallel using user specific logic to produce the output. The output of the map function is passed to shuffle phase.

Shuffle phase receives the output of map functions, sorts them based on keys, and then redistributes data over worker nodes based on the output keys produced by the map functions to ensure that all data belonging to one key is assigned to one worker node for further processing. This is an intermediate phase used to transfer output of map functions to reduce functions.

Reduce phase uses the output of map functions sorted by shuffle phase and usually performs aggregation and group by functions on the input value to generate final output. For a given input of a key and associated list of values $(k_i, v^i_1, v^i_2, \ldots v^i_m)$, a typical reduce function produces a key-value pair in the form (k_i, v_i). All MapReduce cluster nodes run the reduce function in parallel on distinct keys with associated list of values and store the output in the distributed file system.

Hadoop is an open-source implementation of MapReduce programming model written in Java. It is the most commonly used implementation of MapReduce to store and process large datasets in distributed computing environments. Many enterprises including Yahoo, Facebook, and Google use Hadoop for various tasks [11].

Hadoop uses a distributed file system named Hadoop Distributed File System (HDFS) which is a fault tolerant and scalable storage used by Map and Reduce functions to read and write data to achieve the tasks.

A typical Hadoop cluster consists of a master node and multiple worker nodes. The master node is primarily responsible for scheduling and managing jobs. However, worker nodes are used to run map and reduce functions. The HDFS is also installed on the cluster nodes.

2.2 Spark

Spark is a distributed in-memory data processing engine commonly used for batch and stream processing of large datasets to achieve high performance. Spark holds intermediate results in memory rather than writing them to disk which provides near real-time processing of the data and its performance is several times faster than other big data technologies including Hadoop. Spark provides APIs to write applications in Java, Scala, Python, and R with more than 80 high-level operations that help to build distributed and parallel processing systems. Spark provides various modules for machine learning, stream processing, and interacting with SQL-based databases.

Resilient distributed datasets (RDD) is a fundamental data structure used in Spark. It is capable of storing distributed objects of any type. A typical RDD is logically partitioned which which facilitates easy parallel processing on multiple computing nodes offering fault tolerance and ease of use. While RDDs continue to evolve, two operations, namely, *transformation* and *action* can be performed on any RDD.

A transformation operation is used to apply a specific function on the RDD to create a new RDD. A typical example can be a filter method on RDD that returns a new RDD satisfying the filter conditions. Some of the Transformation functions are `map`, `filter`, `flatMap`, `groupByKey`, `reduceByKey`, `aggregateByKey`, `pipe`, and `coalesce`. The output of transformation method is always an RDD.

An action operation evaluates the RDD data and returns a new value. A typical example to use an action on RDD can be finding the number of records in the RDD. Some of the action operations are `reduce`, `collect`, `count`, `first`, `take`, `countByKey`, and `foreach`.

3 Related work

Image processing involves complex and time consuming tasks including edge detection [18] and Hough transform [14]. There have been several efforts to improve the speed and performance of these algorithms. For example, Christos et al. [8] proposed and evaluated FPGA implementation of Canny edge detection to improve performance of the algorithm. The proposed solution shows high throughput to process large number of images without on-chip memory issues. Qian et al. [19] presented a distributed Canny edge detection method which computes the edges within an image based on local distribution of the gradients. The proposed algorithms runs on FPGA and yields better performance compared to the original Canny edge detection algorithm. Some recent work [4] also addresses to improve the Canny edge algorithm to extract text from images.

Some methods have been proposed to improve the performance of Hough transform by exploiting the power of graphics processing units (GPUs) and multi-core processors. For example, Halyo et al. [9] implemented Hough transform on multi-core processors and GPUs and report a speedup gain of three times. Braak et al. [17] achieved a speedup of seven times. Yam-Uicab et al. [20] achieved 20 times speedup gain by implementing Hough transform on GPUs using CUDA programming model, optimization is implemented in parallel using GPU programming, allowing a reduction of total run time and achieving four times better performance than the sequential method. Chen et al. [5] evaluated the Hough algorithm on multi-core processors by distributing the images across processors to achieve high throughput.

Image and video processing is a well-established research area. However, limited contributions are done to develop cloud-services to offer scalable video stream analysis. For example, Ashiq et al. [1] proposed an architecture to perform video stream analysis in cloud computing environment. They integrated MapReduce with OpenCV to process images and showed that the performance of processing images is better than GPUs. Yaseen et al. [21] demonstrated video processing using Hadoop. Swapnil et al. [2] proposed a system to use Hadoop image processing interface (HIPI) [16] for processing images to detect Organic Light Emitting Diode (OLE) centers. The solution runs on Hadoop in parallel and provides high throughput and performance to process a large number of images. Jatmiko et al. [12] use MapReduce framework to detect breast and brain cancer, and tumor from Bio-medical images including magnetic resonance imaging (MRI), diffusion tensor imaging (DTI), and single proton emission computed tomography (SPECT). Wei et al. [10] implement Parallel Processing for Massive remotely sensed data using Hadoop.

Apache Spark [23] is widely used to process large data mainly due to better performance and scalability over Hadoop. Spark is used in various image processing and video analysis tasks. For example, Arthanari et al. [3] proposed a system to identify traffic anomalies using video streams. Jinna et al. [13] used Spark to detect redundant, duplicate, and near-duplicate videos from a large video dataset. They designed a new video similarity measure based on Hough transform and sliding window concepts. The proposed system achieved 5.8 times speedup over existing methods. Rathore et al. [15] design a MapReduce algorithm to detect, monitor, and track vehicles on streets using a network of video cameras. A recent work by Chen et al. [6] proposed a parallel random forest algorithm over Skype for large datasets with high speed and accuracy.

To the best of our knowledge, our proposed work is novel and has not been done before. All of the above-mentioned techniques used Hadoop and Spark for specific applications. However, we propose and evaluate services to preprocess images for edge detection and Hough transform that can be used within any application as preprocessing steps. We propose a completely scalable pipeline for image processing tasks. We evaluate our proposed methods on Hadoop and Spark extensively using a wide variation of images as well as cluster nodes.

4 Canny edge detection and hough transform implementations

We have designed and implemented Canny edge detection and Hough transform algorithms in Hadoop and Spark. We use standalone implementation of these algorithms as a baseline method to compare the performance of Hadoop and Spark implementations. Both of these algorithms are not inherently parallelizable. We have transformed these algorithms into Map and Reduce functions with appropriate key-value pairs as input and output to run on Hadoop to process large datasets in parallel. For Spark implementation, we have transformed Canny edge detector and Hough transform into the Spark echosystem and used RDDs efficiently for parallel processing of large datasets. In this section, we explain the implementation details of standalone, Hadoop, and Spark for Canny edge detection and Hough transform algorithms.

4.1 Canny edge detection

4.1.1 Standalone implementation

Algorithm 1 Standalone Canny Edge Detection

```
    Input: Image $I$, thresholds $\tau_l, \tau_h$
    Output: Canny edge image $E$, gradient information $M, \phi$
 1: procedure CANNY-STANDALONE
 2: Convolve image with Gaussian filter $G$
 3:     $I \leftarrow I \circledast G$
 4: Compute gradients
 5:     for each pixel $(x, y)$ do
 6:         $g_x(x,y) = \frac{1}{2}I(x-1,y) - I(x,y) + \frac{1}{2}I(x+1,y)$
 7:         $g_y(x,y) = \frac{1}{2}I(x,y-1) - I(x,y) + \frac{1}{2}I(x,y+1)$
 8:         $M(x,y) = \sqrt{g_x(x,y)^2 + g_y(x,y)^2}$        ▷ magnitude
 9:         $\phi(x,y) = \tan^{-1}(g_y(x,y)/g_x(x,y))$   ▷ orientation
10:     end for
11: Non-maxima suppression
12:     for each pixel $(x, y)$ do
13:         Find 2 neighbours in direction of gradient $\phi(x, y)$
14:         if $M(x,y) \leq M$(any neighbour) then
15:             $M(x,y) \leftarrow 0$
16:         end if
17:     end for
18: Hysteresis thresholding
19:     for each pixel $(x, y)$ do
20:         if $M(x,y) \geq \tau_h$ then
21:             $E(x,y) \leftarrow 1$                      ▷ edge pixel
22:         else
23:             $E(x,y) \leftarrow 0$                  ▷ non-edge pixel
24:         end if
25:     end for
26:     Recursively find non-edge pixels with $M \geq \tau_l$ and
    with an edge pixel as a neighbour. Mark them as edge
    pixels as well.
27: end procedure
```

A standalone implementation of Canny edge detection algorithm is explained in Algorithm 1 for a given image. After noise removal via Gaussian filtering, the gradient vector and its magnitude and orientation are computed at each pixel. This is followed by a non-maxima suppression step that ensures single pixel thick edges. Finally, hysteresis thresholding is applied using user-defined low and high threshold values. Pixel locations with gradient magnitudes greater than the high threshold are marked as edges. Then all other pixels adjacent to edge pixels and with gradient magnitudes greater than low threshold are recursively marked as edge pixels as well.

4.1.2 MapReduce implementation

Our MapReduce implementation which executes on Apache Hadoop is explained in Algorithms 2. The Map function reads images from HDFS and gets the edges using the same logic presented at the standalone implementation of Canny edge detection and finally update the image file with Canny Edges. The map function runs in parallel to process a large number of images. The output of the Map function is provided input to the Reduce function which reads the edge images sequentially from the HDFS and produces the corresponding edge pixel files.

Algorithm 2 Hadoop/MapReduce implementation for Canny edge detection

```
    Input: lowThreshold, highThreshold, set of input images
    Output: Canny Edge pixel files for the input files
 1: procedure CANNY-MAP(key, value)
 2:             ▷ Key: Name of directory, value: image data
 3:     $cannyImage$ ← using Algorithm 1
 4:     if $cannyImage! = null$ then
 5:         context.write(key, $cannyImage$)
 6:     end if
 7: end procedure
 8:
 9: procedure CANNY-REDUCE(key, value)
10:         ▷ key: HDFS directory path to read edge pixel files,
    value: list of canny images
11:     for img in cannyImages do
12:         pixelFile = hdfs.createNewFile(img.getName()+".pixels")
13:         for i=0 to i<img.getWidth() do
14:             for j=0 to j<img.getHeight() do
15:                 pixelValue ← cannyImage.getPixel(i,j)
16:                 if pixelValue is ON then
17:                     pixelFile.write(i+","+j)
18:                 end if
19:             end for
20:         end for
21:     end for
22: end procedure
```

4.1.3 Spark implementation

Algorithm 3 shows the Spark implementation to perform Canny edge detection on the given image. The algorithm, first reads an image in RRD, second it performs canny-edge detection algorithm over it. Third, it scan over all image to identify white pixels and then write them to a file in HDFS corresponding to the given input image.

Algorithm 3 Spark Canny Implementation

```
    Input: lowThreshold, highThreshold, set of input images
    Output: Canny edge pixel files for the input files
 1: procedure CANNY-SPARK(inputFileName)
 2:     inPath ← Path(hdfs://PATH/images/)
 3:     outPath ← Path(hdfs://PATH/CannyEdges/)
 4:     BufferedImage colorImage
 5:     JavaRDD<BufferedImage> cannyImage
 6:     JavaRDD<Integer, Integer> edgePoints
 7:     colorImage ← readFile(inPath+inputFileName)
 8:     cannyImage ← using Algorithm 1 (colorImage)
 9:     k ← 0
10:     for i=0 to i<cannyImage.getWidth() do
11:         for j=0 to j<cannyImage.getHeight() do
12:             pixelValue ← cannyImage.getPixel(i,j)
13:             if pixelValue is ON then
14:                 Tuple <Integer, Integer> tup
15:                 $tup._1() \leftarrow$ i; $tup._2 \leftarrow$ j
16:                 edgePoints[k] ← tup
17:                 k← k+1
18:             end if
19:         end for
20:     end for
21:     edgePoints.saveAsTextFile(outPath+inputFileName)
22: end procedure
```

4.2 Hough transform implementation

4.2.1 Standalone implementation of hough transform

A typical implementation of Hough transform accepts the edge pixel file consists of edge pixels and identify the geometry objects. We implemented a line detection using Hough transform. Algorithm 4 explains the standalone implementation of the Hough transform. The basic idea is for each edge point x_i, y_i to caste a vote for every possible line with polar parameters (r, θ) that could have passed through it. Our implementation reads a text file containing edge points and for each edge point (x, y), it varies the range of θ to compute the corresponding value of r using the equation: $r = x\cos\theta + y\sin\theta$. A vote is cast for the computed (r, θ) pair by incrementing the current votes in an accumulator array. Locally maximum votes within a 3×3 neighbourhood are retained to avoid duplicated line detections. Finally, the accumulator array is thresholded to obtain lines with significant votes. The (r, θ) pairs for these lines are stored in an output file.

Algorithm 4 Standalone Implementation of Hough Transform.

```
    Input: Canny edge pixel file, threshold $\tau$
    Output: Detected lines
 1: procedure HOUGH-STANDALONE
 2:     Initialize 2D accumulator array Arr with zero votes
 3:     for each edge pixel $(x, y)$ do
 4:         for $\theta = 0 \ldots \pi$ do
 5:             $r \leftarrow x\cos\theta + y\sin\theta$
 6:             Arr$[r, \theta]$ ← Arr$[r, \theta]$+1
 7:         end for
 8:     end for
 9:     Retain local maxima in $3 \times 3$ neighbourhoods of Arr.
10:     Arr cells with more than $\tau$ votes correspond to parameters of detected lines
11: end procedure
```

4.2.2 MapReduce implementation of hough transform

Algorithm 5 explains the MapReduce implementation for Hough transform. In Map function, we compute (r, θ) pairs for the given edge pixel files. Each mapper performs a map function independently over a given edge pixel file and stores the resulting (r, θ) pairs in the HDFS. Each Reduce function receives a list of (r, θ) pairs and simply counts the votes for the pairs and if the votes are higher than the user-defined threshold then the pairs are stored in the corresponding file for the given edge pixel files.

Algorithm 5 Hadoop/MapReduce implementation of Hough transform

```
    Input: threshold, edge pixel file
    Output: rThetaFile
 1: procedure HOUGH-MAP(key,values)
 2:         ▷ key: Name of file; value: file containing pixel point
 3:     for each edge pixel $(x, y)$ in value do
 4:         for $\theta = 0 \ldots \pi$ do
 5:             $r \leftarrow x\cos\theta + y\sin\theta$
 6:             rThetaPair← $r + \theta$        ▷ concatenate $r$ and $\theta$
 7:             return Emit(rThetaPair,1)
 8:         end for
 9:     end for
10: end procedure
11:
12: procedure HOUGH-REDUCE(key,values)
13:         ▷ key: Name of file ; value: list of all rThetaPairs.
14:     votes ← 0
15:     for each $i$ in rThetaPairs do
16:                 ▷ compute count for each unique rThetaPair.
17:         $CM[i] \leftarrow CM[i] + 1$
18:                             ▷ CM is a dictionary datastructure
19:     end for
20:     for each $index\_key$ in $CM$ do
21:                         ▷ $index\_keys$ are unique rThetaPairs
22:         $votes \leftarrow CM[index\_key]$
23:         if $votes > threshold$ then
24:             rThetaFile.write($index\_key$)
25:         end if
26:     end for
27: end procedure
```

4.2.3 Spark implementation of hough

Algorithm 6 explains the Spark implementation for Hough transform. First, it loads a given edge point file into an RDD, and map each pixel (x, y) into parameter (r, θ) using equation $r = x\cos\theta + y\sin\theta$ and emits (r, θ) and 1. Then reduce the emitted data by counting each unique (r, θ) in data, then sort it on the basis of their values, and identify the (r, θ) pairs where votes are higher than the user-defined threshold and finally it is written into HDFS.

Algorithm 6 Spark Hough Implementation

```
    Input: threshold, edge pixel file
    Output: rThetaPair file containing geometry objects
 1: procedure HOUGH-SPARK(inputFileName)
 2:     Input: Canny Edge pixel file
 3:     Output: rTheta pairs file
 4:     JavaRDD<integer,Integer> edgePoints ← null
 5:     JavaRDD<String,Integer> rThetaPair ← null
 6:     k← 0; rTheta← null
 7:     edgePoints ← readFile(inPath+inputFileName) ▷
 8:     for each edge pixel $(x, y)$ in RDD do
 9:         for $\theta = 0 \ldots \pi$ do
10:             $r \leftarrow x \cos\theta + y \sin\theta$
11:             Tuple ¡String, Integer¿ tup
12:             tup._1() ← $r, theta$; tup._2 ← 1
13:             rThetaPair[k] ← tup
14:             k← k+1
15:         end for
16:     end for
17:     rThetaPair.Reduce(key)                    ▷ key:rTheta.
18:     rThetaPair.filter(Threshold)
19:                                               ▷ votes> Threshold
20:     rThetaPair.saveAsTextFile
21:                             (outPath+inputFileName)
22: end procedure
```

5 Experimental analysis

5.1 Description of experiments

We have performed an extensive evaluation of the proposed Hadoop and Spark implementations for Canny edge detection and Hough transform on a cluster consisting of four computing nodes. Each node had a 16 GB physical memory, Octa Core i7 CPU, and 2 TB hard disk. The nodes in the cluster were connected by a Gigabit high-speed network. Hadoop 2.7.2 and Spark 2.2.0 were installed on the cluster. To study the effect of the proposed algorithms, we used HD images (2048 × 1153 pixels) crawled from the Internet to build a dataset for experimental evaluation.

Table 1 summarizes the four experiments that we have performed. Experiment 1 reveals performance on a single Canny edge detection job as the number of computing nodes and the number of images in the job is varied. Comparison is made between the performances of a standalone implementation, a Hadoop implementation and a Spark implementation. Experiment 3 differs from Experiment 1 by allowing multiple jobs and varying the number of concurrent jobs from 1 to 4 and fixing the number of images per job to 10K. Experiments 2 and 4 repeat the same process for the Hough transform. For multiple job experiments (3 and 4), only Spark implementations were profiled since Hadoop was shown to be inferior to Spark in the single job experiments.

5.2 Experiment 1: single canny edge detection job

Figure 3 compares execution times to process varying number of images using standalone, Hadoop, and Spark implementations of Canny edge detection. The Hadoop1, Hadoop2, Hadoop3, and Hadoop4 represent different Hadoop clusters consisting of 1, 2, 3, and 4 machines respectively. Similarly, we also used different cluster sizes for Spark implementation. The standalone implementation is used as a baseline method to compare the proposed implementations. Unsurprisingly, for a small number of images, the performance of standalone implementation is excellent comparing to 1 node Hadoop and Spark (Hadoop1 and Spark1). However, for a large number of images *e.g.*, 10,000, the Hadoop and Spark implementations using multiple nodes significantly outperform the standalone implementation.

We also observed that compared to Hadoop, the Spark implementation scales gracefully by increasing the number of cluster nodes. For example, Figure 3a shows the execution time comparison between Hadoop and standalone implementations. We observe that after 7000 images the four nodes Hadoop cluster (Hadoop4) start outperforming the standalone implementation. However, after 9000 images all configurations of Hadoop outperforms the standalone implementation. Figure 3b shows the execution time comparison of Spark and standalone implementations. We observe that 3 and 4 node Spark clusters (Spark3 and Spark4) always perform better than the standalone implementations. However, for images more than 7000, even a single node Spark cluster (Spark1) also starts performing better than the standalone implementation.

Figure 4 shows the speedup for Canny edge detection on 10,000 images using Hadoop and Spark implementations over standalone implementation. We observed that Spark with all different cluster configurations (number of nodes) gives significantly higher speedup comparing to Hadoop.

Table 1 Description of experiments

Experiment #	Job type	# Jobs	# Concurrent jobs	# Nodes	# Images per job	Comparison
1	Canny	1	1	1–4	$1-10K$	H/S/B
2	Hough	1	1	1–4	$1-10K$	H/S/B
3	Canny	Multiple	1–4	1–4	10K	S/B
4	Hough	Multiple	1–4	1–4	10K	S/B

H Hadoop implementation, *S* Spark implementation, *B* baseline standalone implementation

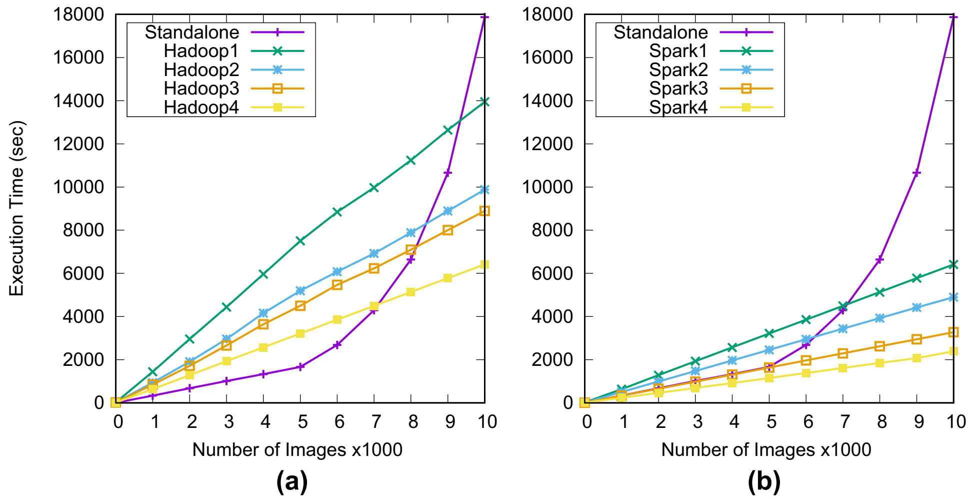


Fig. 3 Experiment 1 (Single Canny Edge Detection Job) comparison of Hadoop (**a**) and Spark (**b**) implementations with the standalone implementation of Canny edge detection for a different number of images. Hadoop1, Hadoop2, Hadoop3, and Hadoop4 shows the number of nodes used in the cluster to profile the execution time. Similarly Spark1, Spark2, Spark3, and Spark4 represent the number of cluster nodes

We obtained the highest speedup of 10.8× using 4 Spark nodes over the standalone implementation. The Hadoop implementation, however, only gives 2.79× speedup over the standalone implementation for Canny edge detection.

5.3 Experiment 2: single Hough transform job

The analysis for a single Hough transform job follows the same patterns as the single Canny edge detection job.

Figure 5 shows a comparison of standalone implementation with Hadoop and Spark implementations for Hough transform on a different number of images. Figure 5a shows the execution time comparison of Hadoop and standalone implementations. We observe that after 5000 images the four nodes Hadoop cluster (Hadoop4) start outperforming the standalone implementation. However, after 9000 images all Hadoop all configurations of Hadoop outperforms the standalone implementation. Figure 5b shows the execution time comparison of Spark and standalone implementations. We observe that 2, 3, and 4 nodes Spark (Spark2, Spark3 and Spark4) always outperform the standalone implementations. However, for images more than 5000, even a single node Spark cluster (Spark1) also outperforms the standalone implementation. Once again, we observed that Spark implementations had better performance than Hadoop as the number of images and cluster nodes was increased.

Figure 6 shows the speedup for Hough transform on 10,000 images using Hadoop and Spark implementations over standalone implementation. We observed that Spark with all different cluster configurations (number of nodes) gives significantly higher speedup compared to Hadoop. We obtained the highest speedup of 9.3× using 4 Spark nodes over the standalone implementation, however, the Hadoop implementation only gives 2.8× speedup over the standalone implementation for Hough transform.

The proposed Canny edge detection and Hough transform using Hadoop and Spark implementations, shown in Experiment 1 and Experiment 2, yield speedup gain over standalone implementations due to automatic data distribution, task scheduling, and high scalability features offered by these frameworks. Moreover, data locality feature of Hadoop helps to reduce the execution time by minimizing data transfer time. The data locality feature ensures to schedule the jobs to the computing nodes hosting the data required by the corresponding tasks which also helps to gain speedup. Whereas, Spark uses resilient distributed dataset (RDD) data structure which loads data in memory to reduce I/O latency and logically divides the data into multiple small chunks. The RDD executes the task on small data chunks in parallel to gain speedup.

A theoretical limit on the maximum possible speedup using Spark for 10,000 images in parallel for Canny edge detection and Hough transform implementations is

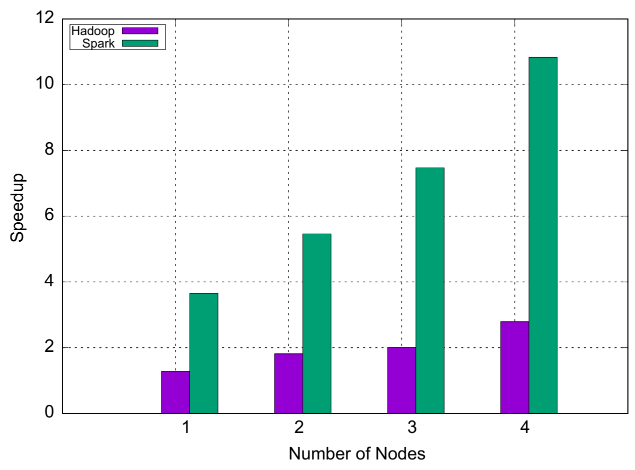


Fig. 4 Experiment 1 (Single Canny Edge Detection Job) speedup for Canny Edge Detection using 10,000 images for both Hadoop and Spark implementations over standalone implementation

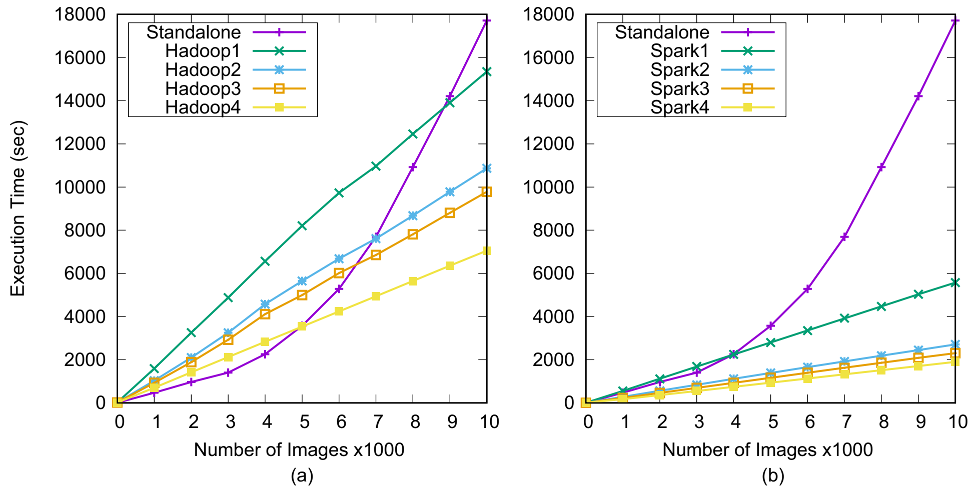


Fig. 5 Experiment 2 (Single Hough Transform Job) comparison of Hadoop (**a**) and Spark (**b**) implementations with the standalone implementation of Hough transform for a different number of images. Hadoop1, Hadoop2, Hadoop3, and Hadoop4 shows the number of nodes used in the cluster to profile the execution time. Similarly Spark1, Spark2, Spark3, and Spark4 represent the number of cluster nodes

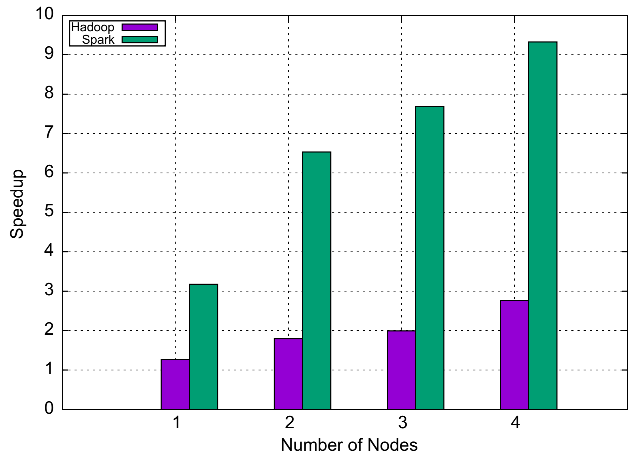


Fig. 6 Experiment 2 (Single Hough Transform Job) speedup for Hough transform using 10,000 images for both Hadoop and Spark implementations over standalone implementation

imposed by Amdahls law $\lim_{p\to\infty} = \frac{1}{f}$, where p is the number of processors, and f is a fraction of these programs which executes in serial and cannot be parallelized. This law shows that a program with a specific f using an infinite number of processors can only give a maximum speedup limit to $\frac{1}{f}$. However, it is challenging to identify the value of f for these programs running in distributed environments like Spark. Fortunately, we can estimate f using Karp–Flatt metric [13]:

$$f = \left(\frac{1}{s_p} - \frac{1}{p}\right) \times \left(1 - \frac{1}{p}\right)^{-1}, \qquad (1)$$

where s_p is the speedup gained using p processors. We used the speedup gained s_p obtained by using 4 nodes over the sequential implementation. In our test bed, each node contains 8 processors, therefore, $p = 4 \times 8 = 32$. We estimate that maximum possible speedup is 15.8× and 12.7× respectively for Canny edge detection and Hough transform using the proposed Spark implementations over a quite larger test bed.

5.4 Experiment 3: multiple concurrent canny edge detection jobs

Table 2 shows the execution time (seconds) for multiple Canny edge detection jobs processed concurrently using a different number of Spark cluster nodes. We variate the number of concurrent jobs from 1 to 4 (1J, 2J, 3J, and 4J) and used different Spark clusters with the number of nodes increasing from 1 to 4 (Spark1, Spark2, Spark3, and Spark4) for processing the Canny edge detection jobs and profile the total execution time. Each job required 10, 000 images to perform Canny edge detection.

We observed that the concurrent number of jobs does not increase the overall execution time significantly as compared to the single job execution time. For example, consider Spark4 for 1J, 2J, 3J, and 4J. The 1J only processes 10,000 images in 1650 s while 4J processes 40,000 images in 1996 s which shows that only 21% additional processing time is required. However, if we process 40,000 images sequentially by processing 1 job of 10,000 images four times in sequence, then we require 300% more

Table 2 Experiment 3 (Multiple Concurrent Canny Edge Detection Jobs) results showing execution time (seconds) for multiple concurrent Canny edge detection jobs on the Spark clusters with different number of nodes

	Spark1	Spark2	Spark3	Spark4
1J	4899	3272	2392	1650
2J	5437	3668	2590	1774
3J	5944	3966	2750	1897
4J	7979	4657	2930	1996

processing time. The performance gain observed in concurrent job processing is mainly due to multiprocessor computing nodes which can serve multiple CPU intensive workload concurrently. Therefore, to exploit and properly utilize the underlying hardware, jobs should be scheduled concurrently.

To show the effect of sequential and concurrent jobs processing on execution time, we consider a case of executing 4 concurrent jobs (4J) on Spark1, Spark2, Spark3, and Spark4 separately and profile the execution time. The sequential job processing time is computed by simply multiplying the time required to process 1 job for the specific Spark configuration with 4. This was compared with the corresponding concurrent job execution time. Figure 7 shows the comparison between four jobs processing sequentially versus concurrently using a different number of Spark node clusters for Canny edge detection. Each job required to process 10,000 images. We observed that Spark implementation gracefully scales on concurrent job processing compared to sequential job execution. We observed $59\%, 64\%, 69\%$, and 70% less execution times to process four concurrent jobs comparing to sequential job execution using 1, 2, 3, and 4 nodes Spark clusters respectively.

5.5 Experiment 4: multiple concurrent hough transform jobs

Table 3 shows the execution time for multiple Hough transform jobs processed concurrently using a different number of Spark cluster nodes. We variate the number of concurrent jobs from 1 to 4 (1J, 2J, 3J, and 4J) and used Spark clusters with the number of nodes increasing from 1 to 4 (Spark1, Spark2, Spark3, and Spark4) for processing

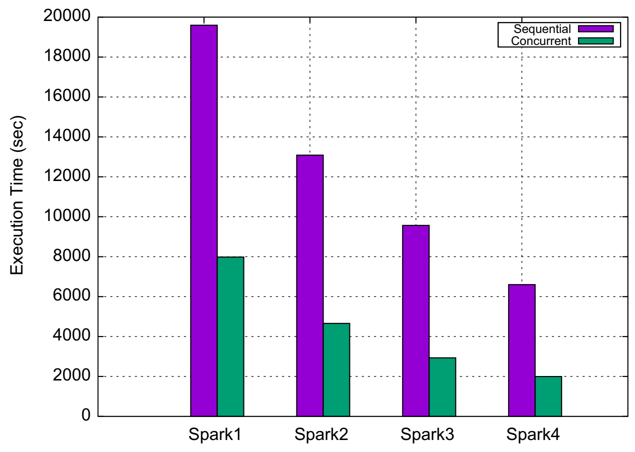


Fig. 7 Comparison of execution time between four jobs (4J) executed sequentially versus concurrently using different number of Spark nodes in Experiment 3 (Multiple Concurrent Canny edge detection Jobs)

Hough transform jobs and profile the total execution time. Each job required 10,000 images to perform the Hough transform.

We observed that the concurrent number of jobs does not increase the overall execution time significantly as compared to the single job execution time. For example, consider Spark4 for 1J, 2J, 3J, and 4J. The 1J only processes 10,000 images in 1900 s while 4J processes 40,000 images in 4115 s which shows that only 116% additional processing time is required. However, if we process 40,000 images sequentially by processing 1 job of 10,000 images four times in sequence, then we require 300% more processing time.

To show the effect of sequential and concurrent jobs processing on execution time, we consider a case of executing 4 concurrent jobs (4J) on Spark1, Spark2, Spark3, and Spark4 separately and profile the execution time. The sequential job processing time is computed by multiplying the time required to process 1 job for the specific Spark configuration by 4 and compared it with the corresponding concurrent job execution time. Figure 8 shows the comparison between four jobs processing sequentially versus concurrently using a different number of Spark node clusters for Hough transform. Each job required to process 10, 000 images. We observed that Spark implementation gracefully scales on the concurrent number of job processing comparing to the sequential job execution. We observed $61\%, 55\%, 50\%$, and 46% less execution time to process four concurrent jobs comparing to sequential job execution time using 1, 2, 3, and 4 nodes Spark clusters respectively.

6 Conclusion

Cloud-based services for image processing are required for automating various tasks. In this paper, we have presented two image processing algorithms, namely Canny edge detection and Hough transform which are complex image processing methods and consume substantial execution

Table 3 Experiment 4 (Multiple Concurrent Hough Transform Jobs) results showing execution time (seconds) for multiple concurrent Hough transform jobs on the Spark clusters with different number of nodes

	Spark1	Spark2	Spark3	Spark4
1J	5577	2712	2306	1900
2J	5867	2848	2795	2741
3J	6937	3749	3535	3245
4J	8689	4849	4561	4115

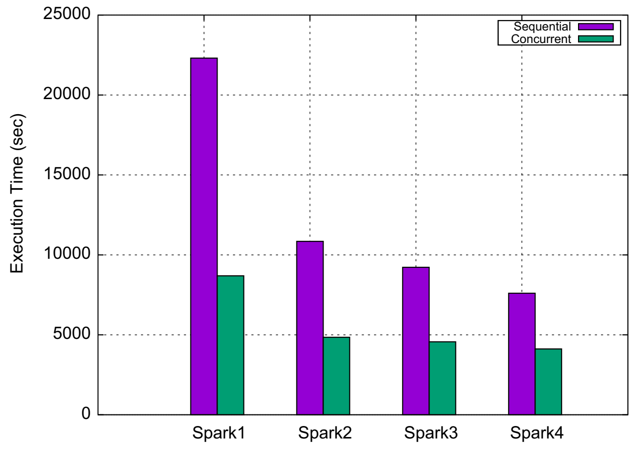


Fig. 8 Spark implementation of Hough transform execution time to process 4 jobs concurrently and sequentially

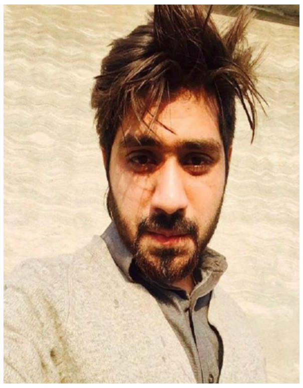
time to process a large number of images. We have presented Hadoop and Spark implementations for both of these algorithms. After extensive experimental evaluation using a different number of images and cluster sizes, we identified that the proposed Spark implementation can provide 10.8× speedup for Canny edge detection and 9.3× speedup for Hough transform to process a large number of images. We also identified that concurrent jobs for Canny edge detection and Hough transform can yield significantly higher performance than processing sequential jobs on the Spark clusters of different sizes.

This work can be extended towards other image and video processing tasks such as keyframe detection, object detection or activity recognition and it can be integrated within a larger, scalable image processing system.

Acknowledgements This work was made possible by NPRP Grant # 7-481-1-088 from the Qatar National Research Fund (a member of Qatar Foundation). The statements made herein are solely the responsibility of the authors.

References

1. Anjum, A., Abdullah, T., Tariq, M., Baltaci, Y., Antonopoulos, N.: Video stream analysis in clouds: an object detection and classification framework for high performance video analytics. IEEE Trans. Cloud Comput. (2016)
2. Arsh, S., Bhatt, A., Kumar, P.: Distributed image processing using hadoop and HIPI. In: 2016 International Conference on Advances in Computing, Communications and Informatics, ICACCI 2016, Jaipur, India, September 21–24, 2016, pp. 2673–2676 (2016)
3. Arthanari, J., Baskaran, R.: Enhancement of video streaming analysis using cluster-computing framework. Clust. Comput. 3 (2018)
4. Arunkumar, P., Shantharajah, S., Geetha, M.: Improved canny detection algorithm for processing and segmenting text from the images. Clust. Comput., pp. 1–7 (2018)
5. Chen, L., Chen, H., Pan, Y., Chen, Y.: A fast efficient parallel Hough transform algorithm on LARPBS. J. Supercomput. **29**(2), 185–195 (2004)
6. Chen, J., Li, K., Tang, Z., Bilal, K., Yu, S., Weng, C., Li, K.: A parallel random forest algorithm for big data in a Spark cloud computing environment. IEEE Trans. Parallel Distrib. Syst. **28**(4), 919–933 (2017)
7. Deanm, J., Ghemawat, S.: MapReduce: simplified data processing on large clusters. Commun. ACM **51**(1), 107–113 (2008)
8. Gentsos, C., Sotiropoulou, C.-L., Nikolaidis, S., Vassiliadis, N.: Real-time canny edge detection parallel implementation for fpgas. In: 2010 17th IEEE International Conference on Electronics, Circuits, and Systems (ICECS), pp. 499–502. IEEE (2010)
9. Halyo, V., LeGresley, P., Lujan, P., Karpusenko, V., Vladimirov, A.: First evaluation of the CPU, GPGPU and MIC architectures for real time particle tracking based on Hough transform at the LHC. J. Instrum. **9**(04), P04005 (2014)
10. Huang, W., Meng, L., Zhang, D., Zhang, W.: In-memory parallel processing of massive remotely sensed data using an Apache Spark on Hadoop YARN model. IEEE J. Sel. Topics Appl. Earth Obs. Remote Sens. **10**(1), 3–19 (2017)
11. Ismail El-Helw, R. H.: Scaling mapreduce vertically and horizontally. In: SC14: International Conference for High Performance Computing, Networking, Storage and Analysis (2014)
12. Jatrniko, W., Arsa, D.M.S., Wisesa, H., Jati, G., Ma'sum, M.A.: A review of big data analytics in the biomedical field. In: International Workshop on Big Data and Information Security (IWBIS), pp. 31–41. IEEE (2016)
13. Lv, J., Wu, B., Yang, S., Jia, B., Qiu, P.: Efficient large scale near-duplicate video detection base on Spark. In: 2016 IEEE International Conference on Big Data (Big Data), pp. 957–962. IEEE (2016)
14. Mukhopadhyay, P., Chaudhuri, B.B.: A survey of hough transform. Pattern Recognit. **48**(3), 993–1010 (2015)
15. Rathore, M.M., Son, H., Ahmad, A., Paul, A., Jeon, G.: Real-time big data stream processing using GPU with Spark over Hadoop ecosystem. Int. J. Parallel Program. pp. 1–17 (2017)
16. Sweeney, C., Liu, L., Arietta, S., Lawrence, J.: HIPI: A Hadoop Image Processing Interface for Image-Based Mapreduce Tasks. University of Virginia, Chris (2011)
17. van den Braak, G.-J., Nugteren, C., Mesman, B., Corporaal, H.: Fast hough transform on gpus: Exploration of algorithm trade-offs. In: International Conference on Advanced Concepts for Intelligent Vision Systems, pp. 611–622. Springer (2011)
18. Waghule, D.R., Ochawar, R.S.: Overview on edge detection methods. In: 2014 International Conference on Electronic Systems, Signal Processing and Computing Technologies (ICESC), pp. 151–155. IEEE (2014)
19. Xu, Q., Varadarajan, S., Chakrabarti, C., Karam, L.J.: A distributed canny edge detector: algorithm and FPGA implementation. IEEE Trans. Image Process. **23**(7), 2944–2960 (2014)
20. Yam-Uicab, R., Lopez-Martinez, J., Trejo-Sanchez, J., Hidalgo-Silva, H., Gonzalez-Segura, S.: A fast hough transform algorithm for straight lines detection in an image using gpu parallel computing with CUDA-C. J. Supercomput. **73**(11), 4823–4842 (2017)
21. Yaseen, M.U., Anjum, A., Rana, O., Hill, R.: Cloud-based scalable object detection and classification in video streams. Future Gener. Comput. Syst. **80**, 286–298 (2018)
22. Zaharia, M., Chowdhury, M., Franklin, M.J., Shenker, S., Stoica, I.: Spark: cluster computing with working sets. HotCloud **10**, 10–10 (2010)
23. Zaharia, M., Xin, R.S., Wendell, P., Das, T., Armbrust, M., Dave, A., Meng, X., Rosen, J., Venkataraman, S., Franklin, M.J., et al.: Apache spark: a unified engine for big data processing. Commun. ACM **59**(11), 56–65 (2016)



Bilal Iqbal received M.Phil and B.S. Software Engineering degree from University of the Punjab, Lahore, Pakistan in 2018 and 2015 respectively. Bilal is currently a machine learning and big data consultant. His research interests include big data, cloud computing, machine learning, and scalable application development.

Waheed Iqbal received the Ph.D. degree in computer science from the Asian Institute of Technology, Bangkok, Thailand, in 2012. He received dual Masters degrees in Computer Science and Information Technology from the Asian Institute of Technology and the Technical University of Catalonia (UPC), Barcelona, Spain, in 2009. Since 2014, he has been an Assistant Professor with the Punjab University College of Information Technology, University of the Punjab, Lahore, Pakistan. His research interests lie in big data, cloud computing, distributed systems, and machine learning.

Nazar Khan is an assistant professor at Punjab University, College of Information Technology, Lahore, Pakistan. He received his B.Sc. degree in computer science from Lahore University of Management Sciences, Pakistan, in 2003, an MSc degree in computer science from the University of Saarland, Germany, in 2007, and a Ph.D. in computer science from the University of Central Florida, USA, in 2013. His research interests are in computer vision and machine learning.

Arif Mahmood is an Associate Professor with the Department of Computer Science, Information Technology University (ITU), Lahore, Pakistan. He received his Masters and the Ph.D. degrees in Computer Science from the Lahore University of Management Sciences in 2003 and 2011 respectively with Gold Medal and academic distinction. He also worked as Postdoc researcher with Qatar University and as Research Assistant Professor with the School of Mathematics and Statistics, and with the school of Computer Science and Software Engineering, the University of the Western Australia (UWA). His major research interests are in Computer Vision and Machine Learning. More specifically he has performed research in data clustering, classification, action and object recognition using image sets, scene background modeling, and person segmentation and action recognition in crowds. He is actively pursuing applications of Machine Learning for the resource management and service quality improvement in cloud computing. Before that he worked on community detection in social and scientific networks for characterizing the network structure.

Abdelkarim Erradi is an Assistant Professor in the Computer Science and Engineering Department at Qatar University. His research and development activities and interests focus on autonomic computing, self-managing systems and cybersecurity. He leads several funded research projects in these areas. He has authored several scientific papers in international conferences and journals. He received his Ph.D. in computer science from the University of New South Wales, Sydney, Australia. Besides his academic experience, he possesses 12 years professional experience as a Designer and a Developer of large scale enterprise applications.