PARALLELIZATION OF SEQUENTIAL COMPUTER VISION ALGORITHMS ON BIG-DATA USING DISTRIBUTED CHUNK-BASED FRAMEWORK

By

Norhan Magdi Sayed Osman

A Thesis Submitted to the
Faculty of Engineering at Cairo University
in Partial Fulfillment of the
Requirements for the Degree of
MASTER OF SCIENCE
in
Electronics and Communications Engineering

FACULTY OF ENGINEERING, CAIRO UNIVERSITY
GIZA, EGYPT
2018

PARALLELIZATION OF SEQUENTIAL COMPUTER VISION ALGORITHMS ON BIG-DATA USING DISTRIBUTED CHUNK-BASED FRAMEWORK

By

Norhan Magdi Sayed Osman

A Thesis Submitted to the
Faculty of Engineering at Cairo University
in Partial Fulfillment of the
Requirements for the Degree of
MASTER OF SCIENCE
in
Electronics and Communications Engineering

Under the Supervision of

Prof. Dr. Hossam Aly Hassan Fahmy
Professor
Electronics and Communications Engineering Department
Faculty of Engineering, Cairo University

Dr. Mohamed Mohamed Rehan
Chief Technology Officer
Avidbeam Technologies

FACULTY OF ENGINEERING, CAIRO UNIVERSITY
GIZA, EGYPT
2018

PARALLELIZATION OF SEQUENTIAL COMPUTER VISION ALGORITHMS ON BIG-DATA USING DISTRIBUTED CHUNK-BASED FRAMEWORK

By

Norhan Magdi Sayed Osman

A Thesis Submitted to the
Faculty of Engineering at Cairo University
in Partial Fulfillment of the
Requirements for the Degree of
MASTER OF SCIENCE
in
Electronics and Communications Engineering

Approved by the Examining Committee:

Prof. Dr. Hossam Aly Hassan Fahmy, Thesis Main Advisor

Dr. Mohamed Mohamed Rehan, Advisor
Chief Technical Officer, AvidBeam Technologies

Prof. Dr. Elsayed Eissa Abdo Hamyed, Internal Examiner

Prof. Dr. Khaled Mostafa Elsayed, External Examiner
Faculty of Computers and Information, Cairo University

FACULTY OF ENGINEERING, CAIRO UNIVERSITY
GIZA, EGYPT
2018

Engineer's Name:	Norhan Magdi Sayed Osman	
Date of Birth:	//	
Nationality:	Egyptian	
E-mail:		
Phone:		
Address:		
Registration Date:	1/3/2014	
Awarding Date:	/ /2018	
Degree:	Master of Science	
Department:	Electronics and Communications Engineering	
Supervisors:		
	Prof. Dr. Hossam Aly Hassan Fahmy	
	Dr. Mohamed Mohamed Rehan	
Examiners:		
	Prof. Dr. Hossam Aly Hassan Fahmy	(Thesis Main advisor)
	Dr. Mohamed Mohamed Rehan	(Advisor)
	Chief Technology Officer, AvidBeam	
	Prof. Dr. Elsayed Eissa Abdo Hamyed	(Internal examiner)
	Prof. Dr. Khaled Mostafa Elsayed	(External examiner)
	Faculty of Computers and Information,	
	Cairo University	

Title of Thesis:

PARALLELIZATION OF SEQUENTIAL COMPUTER VISION ALGORITHMS ON BIG-DATA USING DISTRIBUTED CHUNK-BASED FRAMEWORK

Key Words:

BiG-Data; Computer Vision; Parallel Computing; Video Processing; Video-Chunks

Summary:
In this thesis, we propose a complete framework that enables big-data frameworks to run sequential computer vision algorithms in a scalable and parallel way. Our idea is to divide the input video files into small chunks that can be processed in parallel without affecting the quality of the resulting output. We developed an intelligent data grouping that enables chunk-based framework to distribute data chunks among the available resources and gather the results out of each chunk faster than the standalone sequential approach.

Acknowledgements

I would like to thank my father, my husband and the rest of my family for being patient and supportive during my MSc. period. They believed in me and encouraged me to continue my work with endless love and understanding. I would also like to thank my supervisors, Dr. Hossam Fahmy and Dr. Mohamed Rehan for their continuous support, patience and guidance to me in order to enhance my work and correct my mistakes during this thesis. I really want to thank all my colleagues at AvidBeam who stood beside me to understand several computer vision basics and helped me to mature my developed work. I would like to thank Dina Helal, Menna Ghoneim, Rana Morsi, Ahmed Hegazy, Eslam Ahmed, Mickael Sedrak, Raghda Hassan and Afnan Hassan as well. Special thanks goes to Dr. Hani El Gebaly, Hossam Sami and AvidBeam Technologies for providing support, time and hardware capabilities to develop and test my proposed work and reach the final results.

Dedication

To the soul of my *mother*, I know you are proud of me.

Table of Contents

List of Tables

List of Figures

List of Abbreviations

AM	Application Master
API	Application programming interface
CNN	Convolutional Neural Network
CPU	Central Processing Unit
CV	Computer Vision
DAG	Direct Acyclic Graph
DCB-Framework	Distributed Chunk-Based Framework
DBMS	Database Management System
DSFU	Data Splitting and Feed Unit
DMRMU	Decision Making and Resources Mapping Unit
FIFO	First In First Out
FPS	Frames Per Second
GFS	Googles File System
GPU	Graphyical Prcoessing Unit
GOP	Group of pictures
HDFS	Hadoop Distributed File system
JAR	Java Archive
JVM	Java Virtual Machine
LPR	License Plate Recognition
MPEG	Moving Picture Experts Group
NM	Node Manager
NFS	Network File Systems
OCR	Optical Character Recognition
OpenCV	Open Source Computer Vision Library
OpenMP	Open Multi-Processing
OS	Operating System
PC	Personal Computer
QoS	Quality of Service
RCU	Resources Calculation Unit

RDD	Resilient Distributed Dataset
ROI	Region Of Interest
RTSP	Real Time Streaming Protocol
SIMR	Spark In MapReduce
SQL	Structured Query Language
RM	Resource Manager
UI	User Interface
VM	Virtual Machine
VP	Video Processing
YARN	Yet Another Resource Negotiator

List of Terms

Bolts	Data processing entities at Storm.
Executor	A thread runs at Storm worker process.
Kafka	An open source stream processing tool.
Kestrel	A simple, distributed message queue system
MapReduce	Parallel data processing scheme with two stages: Map and Reduce
Nimbus	A Java daemon runs at Storm master node.
Scheduler	An algorithm used to distribute processing resources on big-data tools
Storm	An open source big-data tool
Spouts	Storm data sources.
Supervisor	A Java daemon runs at Storm worker node
Thrift	A software framework used for scalable cross-language services development
Topology	A directed graph that connects Storm spouts and bolts together.
Zookeeper	An entity that governs the communication between Storm nimbus and supervisors

Abstract

The research presented in this thesis addresses the parallelization of sequential computer vision algorithms, such as motion detection, tracking, etc., on big-data tools. Computer vision sequential algorithms have restrictions on how the video frames should be processed. In these algorithms, the inter-relation between successive frames is important part of the algorithm sequence as the result of processing one video frame depends on the result of the previous processed frame(s).

Most of the present big-data processing frameworks distribute the input data randomly across the available processing units to utilize them efficiently and preserve working load fairness. Therefore, the current big-data frameworks are not suitable for processing video data with inter-frame dependency. When processing these sequential algorithms on big-data tools, splitting the video frames and distributing them on the available cores will not yield the correct output. Consequently, the advantage of the processing sequential algorithms on big-data framework becomes limited only to certain cases where video streams are coming from different input sources.

In this thesis, we propose a complete framework that enables big-data tools to execute sequential computer vision algorithms in a scalable and parallel way with limited modifications. Our main objective is to parallelize the processing operation in order to speed up the required processing time. The main idea is to divide the input big-data video files into small chunks that can be processed in parallel without affecting the quality of the resulting output. We have developed an intelligent data grouping algorithm that distributes these data chunks among the available processing resources and gather the results out of each chunk. A parallelized chunk-based data splitter was used to provide the input data chunks concurrently for parallel processing. Then, our grouping algorithm makes sure that all frames that belong to the same chunk are distributed in order to the associated processing cores.

To evaluate the performance of the developed chunk-based framework, we conducted several experimental tests. We used Apache Storm as our big-data framework for its real-time performance. Storm framework was modified to support input video frame splitting and parallel execution. We examined the behavior of our proposed framework against different number of chunks over one hour testing videos. In our evaluation, we used several sequential computer vision algorithms including face detection, video summarization, license plate recognition (LPR), and heatmaps. Those algorithms were integrated into our testing platform. The results of the chunk-based parallelization achieved a speedup factor from to 2.6x to 7.8x based on the used computer vision algorithm. The processing of multiple video files using different number of chunks was also evaluated using the face detection algorithm in which case, we have achieved up to 8x speedup gain.

Chapter 1: Introduction

1.1 Motivation and Overview

Many of the modern successful business models are built on the idea of gathering specific types of data and analyzing them in order to extract many beneficial results, trends and insights that attract the end user [1]. Useful data types can be in the form of text information, captured images and recorded videos that are gathered offline or online on the real time. To produce accurate and intuitive results, huge amount of data samples must be gathered and processed simultaneously in an efficient way [2]. So, a lot of big-data computation tools were developed to fulfill the need of ingesting and processing big-data in addition to utilizing the available processing resources efficiently.

Video data is one of the fastest growing data types nowadays. This is a result of the ongoing video content that is being created from web users and the various entertainment services being offered to users that are based on video data. Also thanks to the high-end cameras and smartphones that are available for any user anywhere. All of these sources produce an ever growing video content. According to Statista [3], YouTube users upload 400 hours of video content per minute in July 2015. This will be x10 times larger just by 2020 as illustrated in Figure 1.1. Video content is a heavy data type that requires high computational resources for processing. For example, one minute of high quality video requires storage almost as much as 2000 text pages need [4].

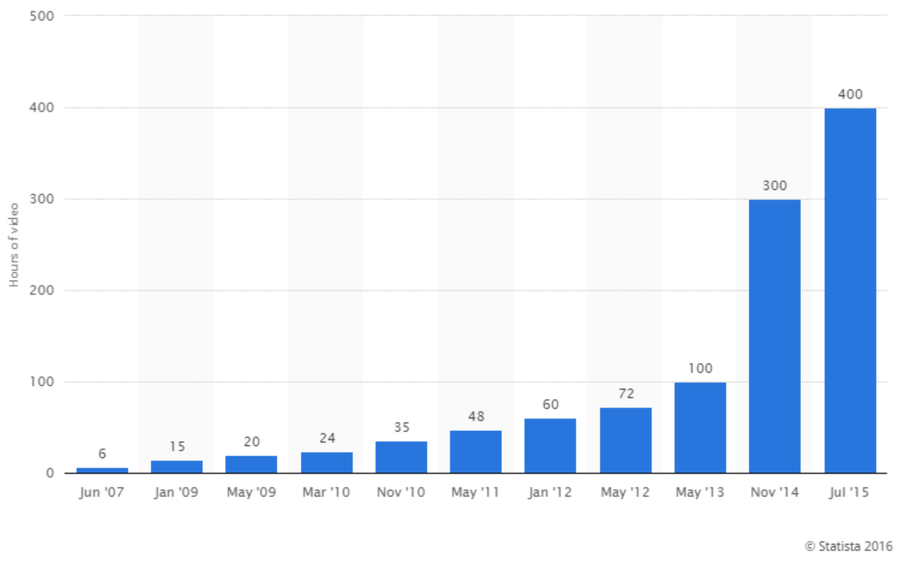


Figure 1.1: Number of hours of uploaded videos to YouTube till July 2015. [4]

Video surveillance systems are currently installed in almost all governmental and private premises as well as highways and malls. Hundreds of hours are being recorded 24/7 to monitor various important locations and detect any wrong or suspicious behaviors [2, 5]. As an example, in case of a crime or an accident, all the recorded surveillance videos can be reviewed to know exactly what happened in a live documented video. The challenge here is that in most cases it is required to analyze these massive amount of data as fast as possible and get the needed results. Many of computer vision algorithms that are applied over these videos are operated sequentially over each video record frame by frame. Examples of these algorithms including video decoding, video summarization, motion detection, object detection and tracking.

In this thesis, Apache Storm will be used as the big-data tool or framework for the evaluation of our proposed framework and produce the necessary validation results. Apache Storm is one of the growing big-data tools that processes vast amount of real time data in distributed clusters. Storm processing framework is called a topology which consists of two important components; spouts and bolts. Spouts are the data sources that receive data streams which need processing. Bolts are the main component of Storm that perform the actual processing operations over the data received by Spouts. Data streams must be distributed between the available bolts in a way that utilize them efficiently and give the best performance.

1.2 Introduction to Big-Data

In this section we will start by discussing the meaning of big-data as a broad term and the dimensions that characterise big-data and define its behaviour. Then we will shed the light over various application fields that need and utilize big-data.

1.2.1 Definition

The Big-Data is a new arising term that refers to the large amount of data that needs to be analysed and processed to extract useful information, trends, insights, patterns and analytics. Traditional data processing software and hardware don't have the ability to handle this massive amount of data. The concept of big-data is not only related to the size and volume of data. Based on Laneys work in 2001 [6] the concept of the triple Vs was introduced to characterize data management based on three dimensions; Volume, Variety and Velocity.

Volume represents data size and magnitude which usually exceed terabytes and petabytes in case of big-data. Variety of data refers to the different types of forms that data can take such as structured, unstructured and semi-structured data [2, 7]. Structured data is the data that has a known format and size and can be stored in relational database while unstructured data is the data that has no defined form and cannot be stored at row-column format [8]. Examples of unstructured data are audio, image files and video recordings. Variety in the sense of big-data is the heterogeneous forms that big-data can take between structured and unstructured data types.

Velocity is the speed of creating new data. Big-data is rapidly generated where almost 90 percent of the worldwide data was created in the past two years [9]. Thanks to the massive amount of web traffic, data collected from sensor networks, different digital transactions, and the enormous numbers of mobile devices that are used worldwide. Such high rates of data generation need fast, efficient and real time analysis and processing that cannot be achieved by traditional data management tools and processing engines.

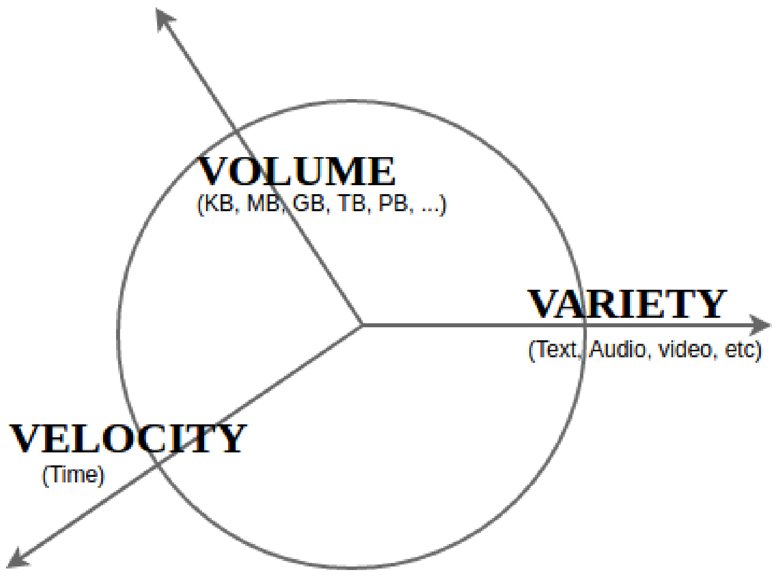


Figure 1.2: Triple Vs of big-data [9]

The triple Vs concept was the base foundation to define *Big-Data* term. Later, it was extended to include another four Vs; Veracity, Variability, Visualization and Value [10]. Veracity means that some of the collected data may contains some uncertain or unreliable data that affects the further data analysis and extracted results [2, 11]. That is why big-data faces the challenge of dealing with non-correct or uncertain group of information which requires special tools and algorithms to deal with this veracity nature especially when decisions are taken automatically without human intervention. Variability in big-data means that data comes in different flow rates from various sources [2] and in another definition it means that despite having big-data in the same data form, it may be constructed with different semantics [11].

The third new V refers to Visualization which is a crucial challenge for big-data. Visualization means that large amount of data needs to be presented in an intuitive way to include all the related parameters and to understand the insights and trends through complex graphs yet clear and neat. Value is the last V of big-data dimensions. Such large flood of data is considered to be low density in its original form. The value of big-data comes from all the analytics performed over it and the useful information resulted from this analysis. Many organizations and new startup companies are established over the insights and knowledge gained from processing big-data [10]. This concludes that the future of technology and business will be determined by the power of utilizing big-data and transforming it into beneficial knowledge.

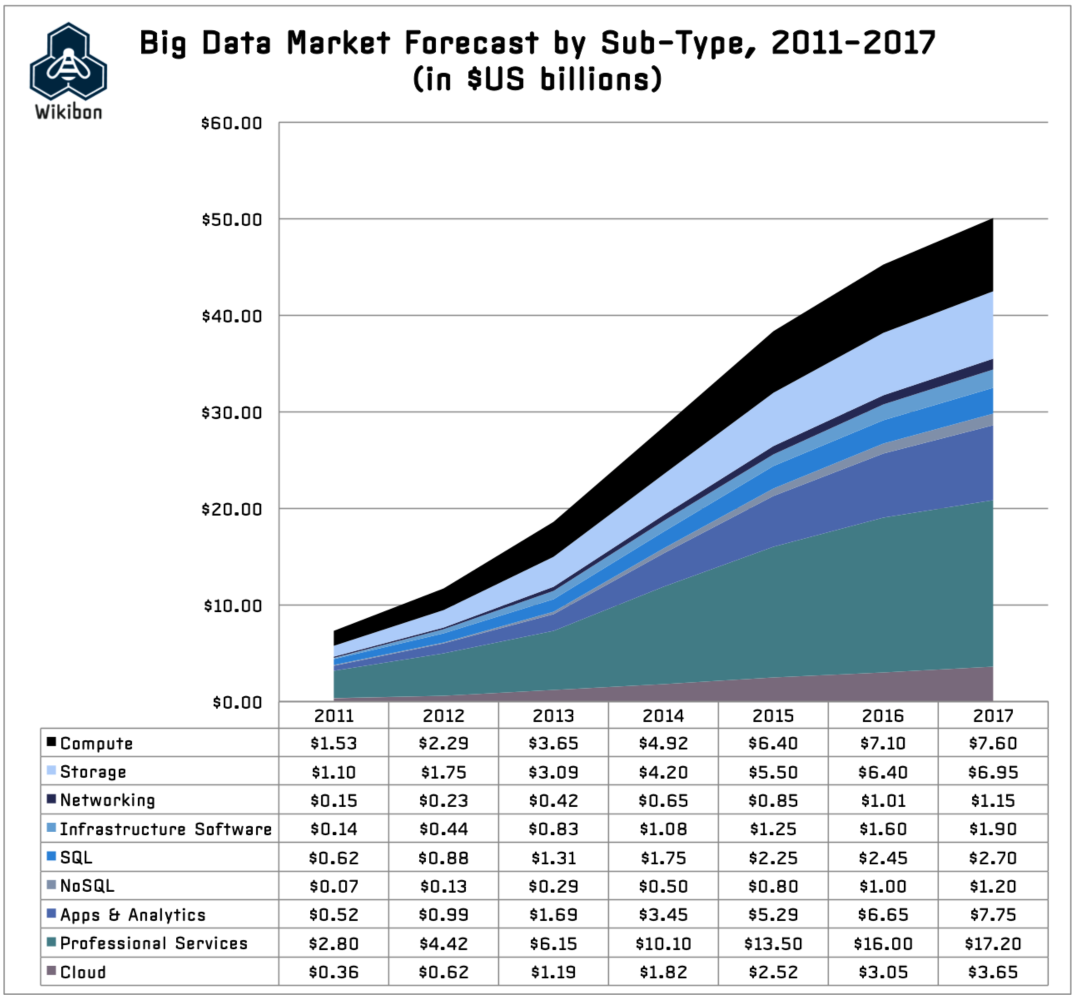
1.2.2 Applications

Big-data has many applications in all arenas from health care, financial transactions, marketing into education and social life. A graphical representation of big-data market forecast by WikiBon is shown in Figure 1.3 [12]. For example, in health care field many new wearable gadgets are used to measure biological signals from human bodies such as heart rate and blood pressure. By collecting such information from large number of people and analyzing it through big-data tools, doctors can have a clear view over many diseases symptoms and how to diagnose them correctly [13].

Mobile devices that have different sensors and distributed in almost everywhere are a valuable source for big-data. Governmental or academic associations can benefit from such tremendous amount of collected sensors data for applicable or research purposes related to marketing, social studies, environmental issues, education, security, enhancing lifestyle and much more.

Another important application for big-data is in social media where many websites such as Facebook, Twitter, Instagram and Flickr are good examples of how big-data can be used to understand how people interact with each other. The large collected people's personal data reveals the society direction towards general topics and how they value some commercial products or artworks. This also can help products manufacturers to know which brands are preferred by the consumers and how to enhance them according to tracking positive or negative sentiment of users feedback over social media [14].

When it comes to security, big-data plays a vital role in taking security decisions, detecting hazards and eliminating dangerous situations. In banking and insurances sectors, big-data analysis techniques help to detect frauds and malformed financial transactions. This is done by analysing huge amount of customers transactions and detect suspicious patterns automatically [15]. Another important type of big-data security applications is obtained from the huge amount of surveillance videos that are recorded 24 hours daily in almost all private and governmental facilities. By analysing these records, we can detect motions, intruders, suspicious behaviours and do further analysis to count people, detect faces, track objects and much more. Recent big-data technologies help to convert unstructured video data into searchable content that can be used for security purposes.



	2011	2012	2013	2014	2015	2016	2017
■ Compute	$1.53	$2.29	$3.65	$4.92	$6.40	$7.10	$7.60
■ Storage	$1.10	$1.75	$3.09	$4.20	$5.50	$6.40	$6.95
■ Networking	$0.15	$0.23	$0.42	$0.65	$0.85	$1.01	$1.15
■ Infrastructure Software	$0.14	$0.44	$0.83	$1.08	$1.25	$1.60	$1.90
■ SQL	$0.62	$0.88	$1.31	$1.75	$2.25	$2.45	$2.70
■ NoSQL	$0.07	$0.13	$0.29	$0.50	$0.80	$1.00	$1.20
■ Apps & Analytics	$0.52	$0.99	$1.69	$3.45	$5.29	$6.65	$7.75
■ Professional Services	$2.80	$4.42	$6.15	$10.10	$13.50	$16.00	$17.20
■ Cloud	$0.36	$0.62	$1.19	$1.82	$2.52	$3.05	$3.65

Figure 1.3: Wikibons Big-data market forecast [12]

1.3 Problem Description

Most of the existing big-data tools such as Hadoop, Spark and Storm process data in parallel by splitting data into small parts and forward each part into a separate processing instance, (Map, core, etc.) for processing. Those processing instances produce intermediate results. The intermediate results produced by all processing units are then aggregated in a later stage using a reducer instance.

This approach is not suitable for several computer vision algorithms where the frames needed to be processed in sequential order. Example of these algorithms includes motion detection algorithms and tracking algorithms.

In motion detection algorithm, each frame is subtracted from the immediately previous frame using background subtraction approach. The result of the subtraction, difference frame, is then analyzed for non-zero data which is used as an indication of motion. In tracking algorithm, once an object is detected, the area surrounding this object is recorded and in the following frame, a matching between the recorded area and the area close to it in the spatial position is computed. The closed match is then considered the new location of the object being tracked.

The two scenarios mentioned above require frames in successive order to be sent to the same processing node which is not the exact case in current big-data solutions. In current big-data solutions, each new frame is being assigned to an empty core randomly so sequential frames will be processed in parallel rather than in sequential order.

As an intermediate solution, we are proposing a new operation mode *Chunk-Based*. In our proposed chunk-based processing, incoming frames are split into a group of chunks. Each chunk has frames in sequential order. By doing this, the algorithms mentioned above can be applied with certain limitation. The limitation of the proposed implementation is due to the cross chunk relations. For example, tracking across chunk border could fail. This limitation can be handled by the topology reducer to eliminate redundant results. However, the performance gain due to splitting the input into many chunks could be dramatic (proportion to number of parallel cores used in the topology). Algorithms such as object detection, tracking and summarization need to process the video frames one by one in order and the results between these frames are dependent. So, depending on the execution speed of the algorithm, video length and video content; we need relatively long duration to examine each video file sequentially and produce the results.

Running sequential computer vision algorithm and video processing as a standalone application has many drawbacks:

1. The execution time is high, almost equals to the video file duration as it will decode each frame individually then process it sequentially in order.
2. The processing of many video files will be manually one file at a time because the algorithm itself cannot handle processing many files in parallel. We have to wait for one file to finish processing then run the algorithm over the next one and so on.
3. Based on the previous point, there will be under utilization to any available processing resources as we are limited by running one file at a time.

1.4 Challenges

According to the importance of using such vital video processing algorithms and the need to parallelize the operation over huge amount of data while using the available resources efficiently, the need arises to use big-data tools to run such applications efficiently. Using a distributed big-data tool such as Storm to run sequential applications over video data has many challenges:

1. Data Feed: Data feed must be adequate to data type nature and how it is required to be processed. In our case, we should know if we need to process video files as a whole or frame by frame and how can Storm accommodate this.
2. Data Distribution: Storm distributes data among the processing bolts in a random way to ensure fairness between resources, this opposes the idea of sequential applications which need data to be processed in order.
3. Data Consistency: Sequential algorithms also need all data from same data source to be processed at the same instance which is not applied by default when using Storm. Storm default data grouping sends data to processing units based on working load and does not consider any other priority or ordering factors.

1.5 Thesis Organization

This thesis is organized as follows: Chapter 1 is the introduction which includes an overview about Big-Data concept. Chapter 2 provides a detailed review about three of the most powerful big-data processing technologies: Apache Hadoop, Apache Spark and Apache Storm. Chapter 3 goes through the literature of Computer Vision and discusses some of the previous work done to parallelize processing of Computer Vision Algorithms and similar to this thesis objectives. Chapter 4 contains the thesis proposed distributed chunk-based processing framework concept and system architecture along with the entire system component functions and workflow. Chapter 5 has the evaluation process methodology conducted to examine the performance of the proposed framework and highlights the speedup gain achieved. Finally, Chapter 6 concludes all the work done within this thesis and suggests new ideas to extend the proposed work.

Chapter 2: Big-Data Technologies

2.1 Introduction

The importance of big-data comes from the valuable information that could be revealed out of such huge and complex data after applying specific analytical procedures and using robust and scalable processing tools. Emerging technologies are being developed to cover all the stages of handling big-data. Big-data technologies target the implementation of all the complex components needed for big-data storing, processing, visualization and producing final results in the required formats as shown in Figure 2.1.

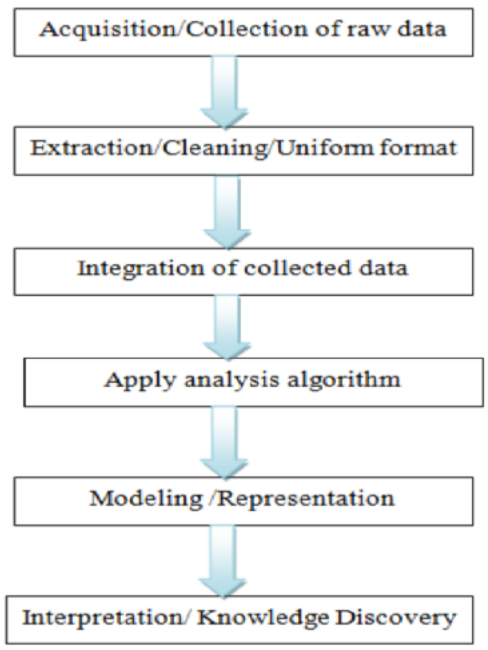

Acquisition/Collection of raw data
Extraction/Cleaning/Uniform format
Integration of collected data
Apply analysis algorithm
Modeling /Representation
Interpretation/ Knowledge Discovery

Figure 2.1: Big-data processing steps [11].

Big-data can be processed in two modes; *Batch Processing* or *Stream Processing*. In batch processing, a group of data is first stored then fed to the big-data tools to be processed at once while in stream processing data is ingested into processing tool straight forward in real-time [16]. Batch processing introduced the concept of MapReduce that allows performing complex computations over big-data in parallel scheme. MapReduce programming model is divided into two main functions; Map which performs the basic processing functions over pairs of key-value inputs, then Reduce which groups all the intermediate results produced by map function and aggregates them into meaningful output

[17]. Big-data processing tools key feature is the ability to perform batch processing, stream processing or both of them. Using big-data technologies enables parallel data processing over a cluster of nodes in an efficient and dynamic way that couldn't be achieved using traditional data handling frameworks.

In this chapter we will talk about three of the most popular big-data processing tools that are designed to cope with the aforementioned big-data characteristics. The studied tools are Apache Hadoop, Apache Spark and Apache Storm. A comprehensive elaboration is conducted to understand the basic concept of each tool and how it works. Then we will conclude to the final tool that will be used in this thesis to achieve parallel processing for sequential computer vision algorithms in a scalable way.

2.2 Apache Hadoop

2.2.1 Overview

Apache Hadoop [18] is a project launched in 2008 by Doug Cutting and Mike Cafarella in Yahoo and University of Michigan [17] . Hadoop was inspired by Google's File System (GFS) from published paper in 2003 that defines GFS and its functions to handle massive amount of data. Hadoop definition by Apache states that it is a software framework that enables the processing of large amount of data over a distributed cluster of commodity hardware in a simple way and failures prone [19]. Moreover Hadoop is an open source framework that is able to store, process and manage big-data in a scalable way.

Hadoop became a suitable option for big-data processing due to several key features. Hadoop cluster consists of large number of cluster nodes. It can scale up or down automatically by adding or removing nodes from the cluster. This made it necessary for Hadoop to be fault tolerant. When one node fails, Hadoop can continue working without any interruption due to data replication across the cluster. Another important feature that Hadoop introduced is bringing the complex computation operations into data storage location rather than moving the data to the computation units. This helped to parallelize the computations and perform them in local nodes [19]. Also Hadoop does not require specific data formats and can be used to process both structured and unstructured data. Hadoop supports batch processing mode. Finally, a core competency for Hadoop is its low running cost where no need for high-end expensive servers and commodity hardware is just enough.

2.2.2 Components and Cluster Architecture

Apache Hadoop cluster consists of three main components; Hadoop Distributed File system (HDFS) [20] for data storage , MapReduce [21] for data analysis and Yet Another Resource Negotiator (YARN) [22] for management.

HDFS is an open source distributed data storage that is designed to store data using commodity hardware in an efficient and redundant way while achieving high throughput. HDFS from architectural point of view is a cluster consists of one master machine named *NameNode* and several distributed slave machines known as *DataNodes* as depicted in Figure 2.2. Data are split into small blocks and stored in data nodes. This makes data nodes responsible for data read and write functions while maintaining multiple copies of data across different data nodes to support fault tolerance [19]. Data are replicated across HDFS cluster using a replication factor which is commonly three [20] and clients usually read from the nearest node.

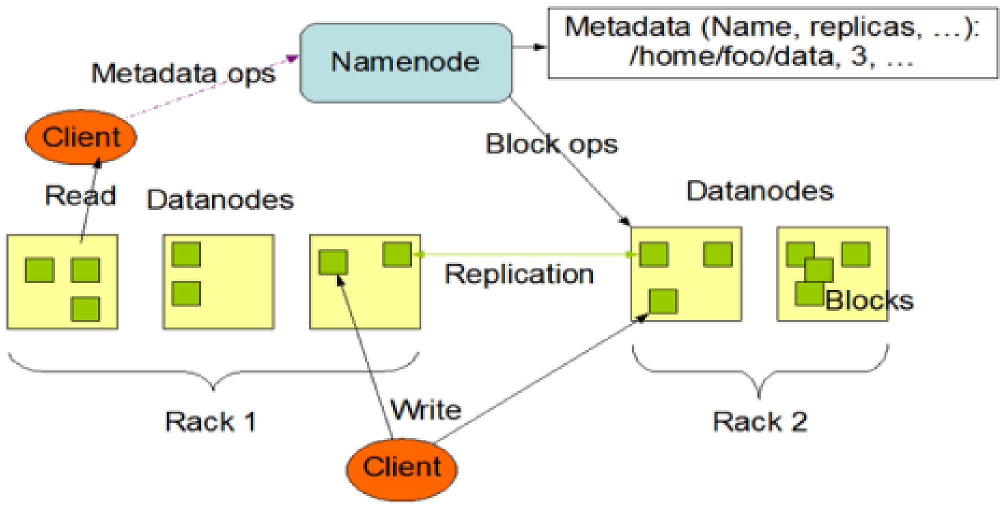


Figure 2.2: HDFS architecture with data replication [19].

NameNode controls data distribution among Hadoop cluster and keeps track of data blocks mapping between cluster data nodes . When a client needs to read data, it sends a request to NameNode which responds with DataNode address and metadata that hosts the required block of data [17]. To make HDFS more robust, a secondary NameNode is used to backup the original NameNode in case of failure. This solves common HDFS high availability issues. HDFS programming is written using Java language. So, any commodity server that runs Linux is a good candidate to be a NameNode or a DataNode. An abstract master node and slave node internal architecture is shown in Figure 2.3.

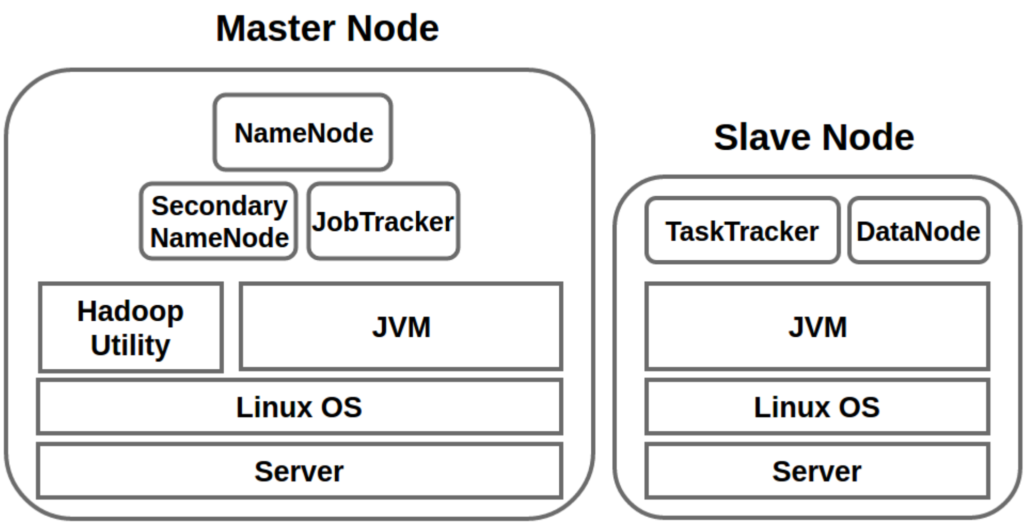


Figure 2.3: HDFS abstract architecture of master and slave nodes.

MapReduce is the part that is responsible for data analysis and processing in Apache Hadoop. It enables Hadoop to perform distributed processing over large datasets. Given that the data is distributed among many working machines, mapreduce framework performs parallel processing over these machines locally where the data resides [19]. The input data is split into small batches that are independent and can be processed in parallel.

MapReduce is divided into two functions: *Map* and *Reduce* that developers can program them in a custom way based on the required application. In Hadoop, the map function is invoked over data batches in different commodity machines distributed within Hadoop cluster where inputs are given as key-value pairs. Data mappers are used to take raw data then convert them into input key-value pairs for map function. The map function executes computations over local data to produce intermediate results that are given to the reduce function as its input. The reduce function uses a custom logic to aggregate the map results, as key-value pairs, then create the final outputs [21]. Figure 2.4 depicts the working flow of Hadoop mapreduce. Hadoop inputs and outputs are read from HDFS and written in HDFS as well using specific input formats.

JobTracker and *TaskTracker* are the two daemons of Hadoop mapreduce. JobTracker daemon is located in NameNode which is the cluster master machine. It submits and tracks mapreduce functions by receiving job requests from client, identifying data location in datanodes and submitting work to the TaskTracker that is located in the chosen DataNode. TaskTracker is located in slave machine. It accepts map or reduce job from JobTracker and assigns it a free working slot then notifies back the JobTracker with the assigned slot and the job status. TaskTracker always reports the JobTracker with number of free slots that are available for new jobs.

A heartbeat signal is maintained between TaskTracker and JobTracker to periodically report the TaskTracker availability. MapReduce recovers any failures using its daemons where any task that fails to successfully send acknowledgement, it is considered failed. Hence, the Tasktracker reports the failed task to JobTracker which reschedules it back to different TaskTracker rather than the one it failed previously within [17, 23]. A simple Hadoop flow using one master node and three slave nodes is shown in Figure 2.5, which illustrates the HDFS and MapReduce basic components and functions in Hadoop Cluster [24]. Although mapreduce was a big leap in parallelizing the processing over vast amount of data, it has some performance limitations. The scalability limit of mapreduce is 4000 node besides it cannot achieve the best resource sharing between different computation frameworks and also handling jobs' working flow is centralized in one JobTracker node [22, 25]. That's why a new framework called YARN was introduced.

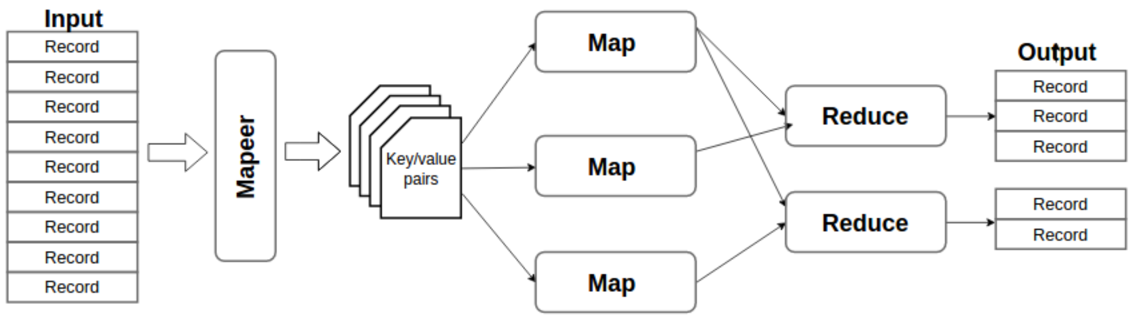


Figure 2.4: Hadoop mapreduce.

YARN is the new generation of mapreduce which aims to solve the limitations of the old framework. YARN was designed to provide the needed computational resources such as CPU and memory for processing jobs. Main components of YARN are *Resource-Manager*, *ApplicationMaster* and *NodeManager*. A ResourceManager is analogous to JobTracker while NodeManager is analogous to TaskTracker in mapreduce [17]. In designing YARN, the JobTracker functionalities were divided between two new components; global *ResourceManager* (**RM**) and per-application *ApplicationMaster* (**AM**). RM is contained in master node and it targets the distribution and tracking of the cluster containers (i.e. resources) over the running applications. The RM has two main components; *Scheduler* which assigns resources to working nodes and *ApplicationManager* which is responsible for job scheduling and restarting the AM in case of failure [25].

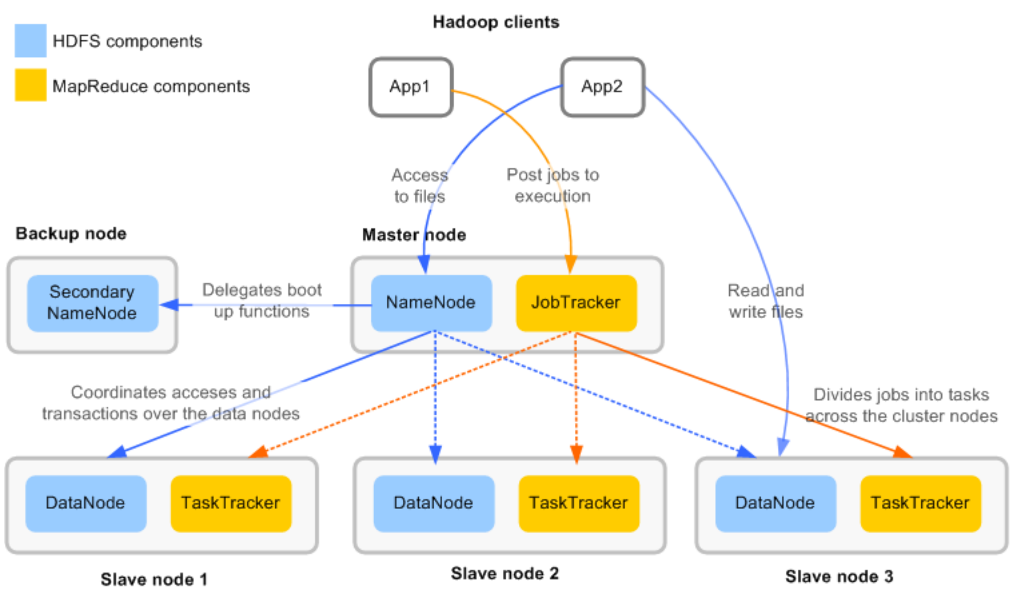


Figure 2.5: Hadoop simple working flow [24].

AM takes charge of executing single application based on Scheduler decisions. The RM negotiates the needed resources to run specific application with the AM then allocates the demanded containers. There is a regular heartbeat message sent from AM to RM to reports its liveness and define its requirements [22]. The *NodeManager* (**NM**) is the worker daemon that is located in every slave machine in the cluster. It is supposed to inform the RM with its node computational capacity and available containers that is why a container is considered to be part of NM capacity in terms of memory and cores. NM initiates or kills working components upon RM requests and execution status.

The workflow of YARN is shown in Figure 2.6:

1. First the client sends application requests using JobSubmitter to RM.
2. The RM asks its Scheduler to allocate a container for the submitted application.
3. The RM asks the associated NM for the requested container.
4. The NM initiates the computational container with the required resources.
5. Finally the AM is run by the container.

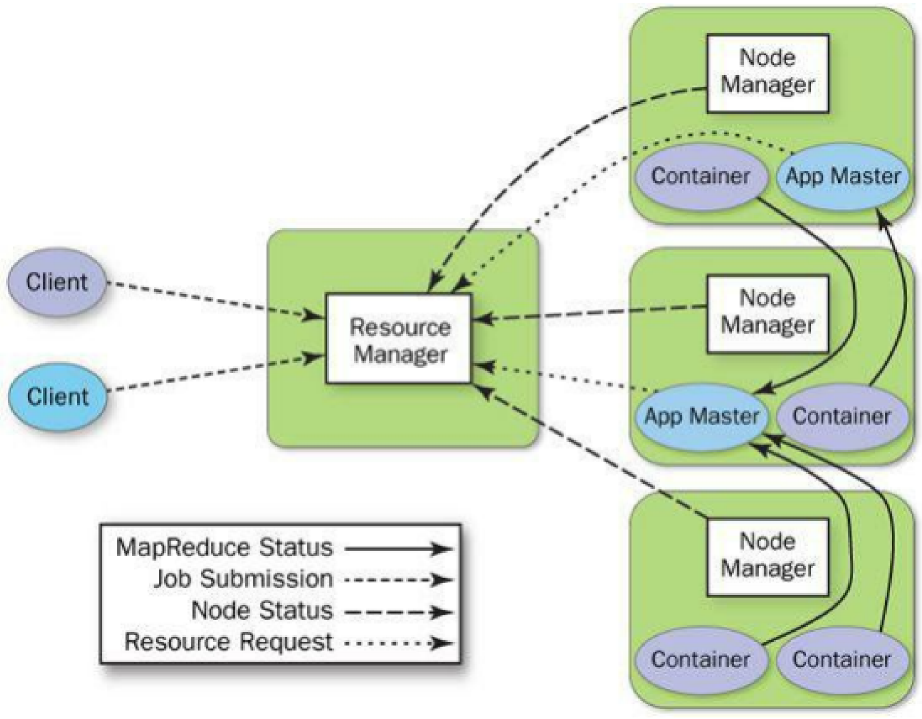


Figure 2.6: Hadoop YARN architecture [26].

2.2.3 Scheduler

Hadoop has different job scheduling techniques. Schedulers basic categories are Static and Dynamic schedulers. In Static Scheduler, resources are allocated for each job before the job starts while in dynamic scheduler the resources assignment decisions are taken during the runtime to change the resources allocation. Examples of Static schedulers are First In First Out (FIFO) and Capacity scheduler while examples of dynamic scheduler are Resource Aware and Deadline Constraint. The default scheduler that is used with JobTracker is FIFO. In FIFO, the first job submitted to JobTracker queue is pulled for work. Another scheduler is Fair scheduler which tries to assign the same average amount of resources for each submitted job over time. Fair scheduler was introduced by Facebook and it is close to Capacity scheduler which was developed by Yahoo except that capacity scheduler is much suitable for large clusters [27].

2.2.4 Limitations

As a big-data processing tool, Hadoop has some limitations that hinder it from giving the ultimate performance. Here we will list some of Hadoop general drawbacks that don't make it a good candidate for big-data video processing [23, 28]:

- Data replication in HDFS isn't efficient in case of big-data, specially video files.
- Master node for HDFS and MapReduce is a single node of failure which risks losing huge and important data in case of master drop.
- HDFS has no optimization for queries that leads to inefficient query execution with high cost and large clusters.
- Hadoop by default does not offer security mechanisms due to its complexity. That's why it isn't suitable for surveillance and security applications that are built for big-data video applications.
- Also no storage or network encryption offered by Hadoop.
- MapReduce framework does not support complex execution models that require different levels of map or reduce or custom functions.
- Hadoop does not cache intermediate data results or store them in databases, although these results may generate vital insights and trends about the original input data.
- Hadoop only supports batch processing, so it cannot handle real-time applications or live video streams.
- Processing speed in Hadoop is relatively low as a result of the long execution time and high latency of mapreduce framework.

2.3 Apache Spark

2.3.1 Overview

Apache Spark [29] is an open source in-memory big-data processing tool that was first developed at UC Berkeley in 2009 [17]. The motivation to create Spark was to design a unified framework for big-data distributed processing. Spark came to solve the limitations of Hadoop and mapreduce while being faster and general in-memory processing engine. Spark utilization of Hadoop is only by using HDFS as storage if needed. Apache Spark is not replacement for Hadoop itself, it only replaces mapreduce in order to perform real-time stream processing and fast interactive queries in no time [30].

Spark is known for its processing speed, it is claimed that Spark can work 100x faster than Hadoop in case of retrieving data from memory and 10x faster for disk case as mentioned in Spark official website [29]. Many programming languages are supported using Spark. Developers using Java, Python or R can effortless write or integrate applications in Spark using different APIs which is an advantage for making Spark easy to use. Other important features of Spark are supporting advanced analytics, SQL streaming, machine learning and graph algorithms [30].

2.3.2 Components and Cluster Architecture

The core of distributed processing for Spark is the Resilient Distributed Dataset (RDD). RDDs are immutable collection of objects, data structures, that are split into small partitions and can be processed in parallel across Spark cluster machines while maintaining data locality [31]. Spark RDDs are read-only, so any intermediate results can be persisted into memory or disk to be used later if needed. Each RDD has a lineage that is used to track and reconstruct any lost RDD. This makes RDD fault tolerant [32]. Originally RDDs are Scala objects that can be constructed from:

- Reading data file from HDFS,
- Parallelizing a Scala collection by slicing it into small partitions to be processed in parallel,
- Transforming an existing RDDs using Spark specific functions. For example Map, Filter and GroupBy functions,
- And finally changing RDDs persistence either by caching or saving them at HDFS for later use.

Spark deals with data streams in micro-batching that splits data into parts and process each part individually. This approach called DStream which represents a queue of RDDs. DStream enables Spark to do both batch and streaming processing in a quick way as long as the application does not require low latency [31]. Processing mode in Spark takes the form of Direct Acyclic Graph (DAG) which is a sequence of direct computations to be executed over data. As shown in Figure 2.7, DAG nodes are RDDs and edges are the transformation functions performed on this data. Data transformation in DAG are acyclic so that the graph goes forward and does not go back to previous step.

Apache Spark has different types of deployments as shown in Figure 2.8. It can be deployed as a *Standalone* framework on top of HDFS and side by side with mapreduce. Or it can run using *Hadoop YARN* as a resource management framework to coexist with Hadoop ecosystem. Another deployment is *Spark In MapReduce* (SIMR) where mapreduce is used to start Spark jobs [30].

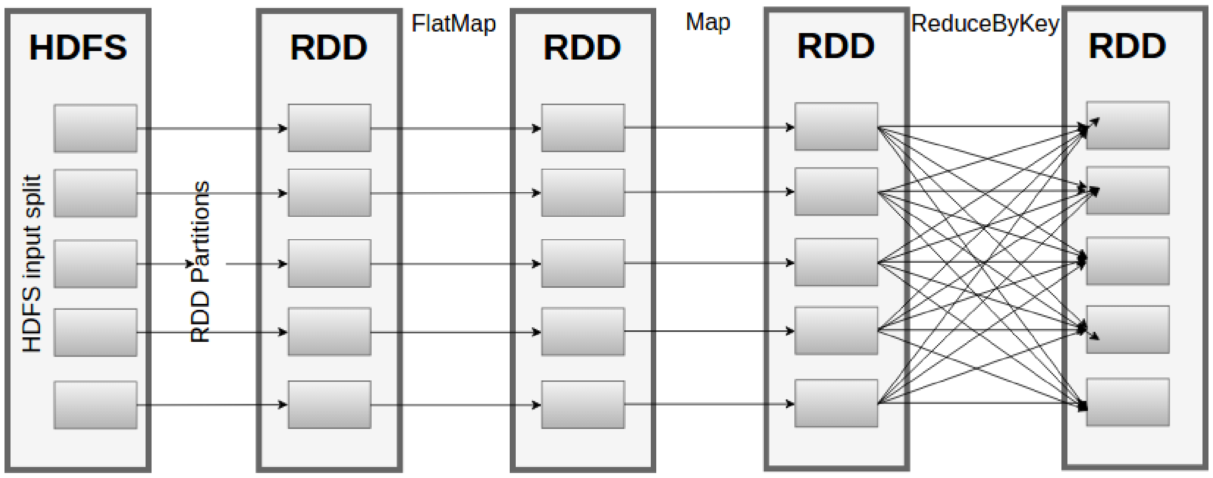


Figure 2.7: Spark DAG.

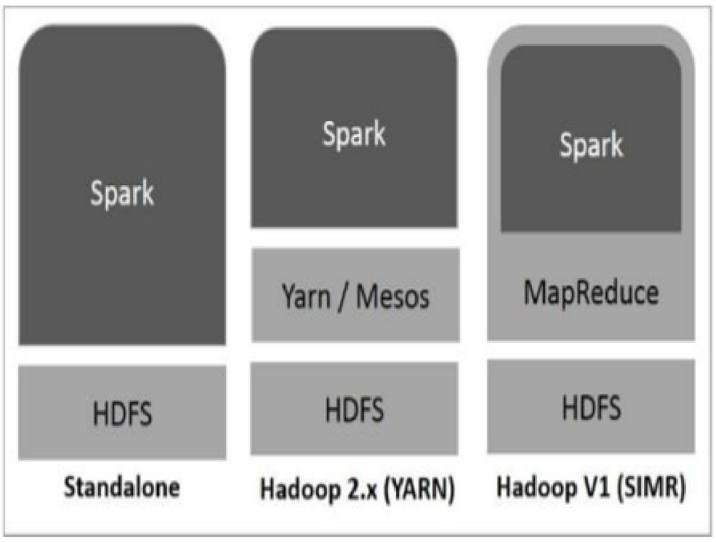


Figure 2.8: Spark different deployment schemes [30].

On top of Spark core, RDD and DAG, a group of high-level implemented components are added to Spark stack as depicted in Figure 2.9. SparkSQL is the component that enables Spark to process structured and unstructured data. Spark Streaming enables Spark to process RDDs streams in mini-batches based on new concept called *discretized streams*. GraphX is a visualization tool that offers usable APIs to express user defined computations [33]. MLib is a popular scalable machine learning library that is built especially for Spark and outperforms Hadoop machine learning engine Apache Mahout [30].

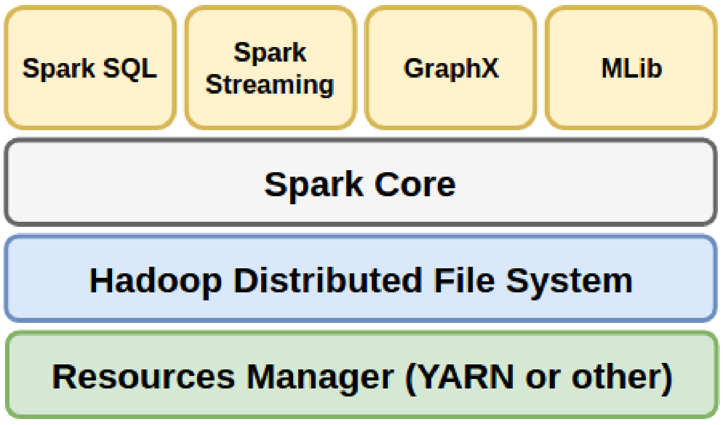

Spark SQL
Spark Streaming
GraphX
MLib
Spark Core
Hadoop Distributed File System
Resources Manager (YARN or other)

Figure 2.9: Spark components.

Spark cluster architecture consists of a master node and a group of slave nodes. A Driver node/Driver manager is a JVM that creates SparkContext for Spark application, transforms RDDs into DAG, stores metadata about RDDs and schedules Spark Tasks [34]. Driver monitors DAGScheduler and TaskScheduler. Spark Application is divided into small tasks that are spawned to Spark executors within worker nodes governed by TaskScheduler. Driver Manager is similar to ApplicationManager in Hadoop, it also communicates with an entity called Cluster Manager in master node which governs the cluster resources management. Worker Nodes are where the actual data processing is done [35]. One worker node has multiple executors. Spark Executor caches data or intermediate results, reads/writes data from/to external source and performs data processing [34]. From Figure 2.10 we can have a glimpse view about Spark abstract cluster architecture.

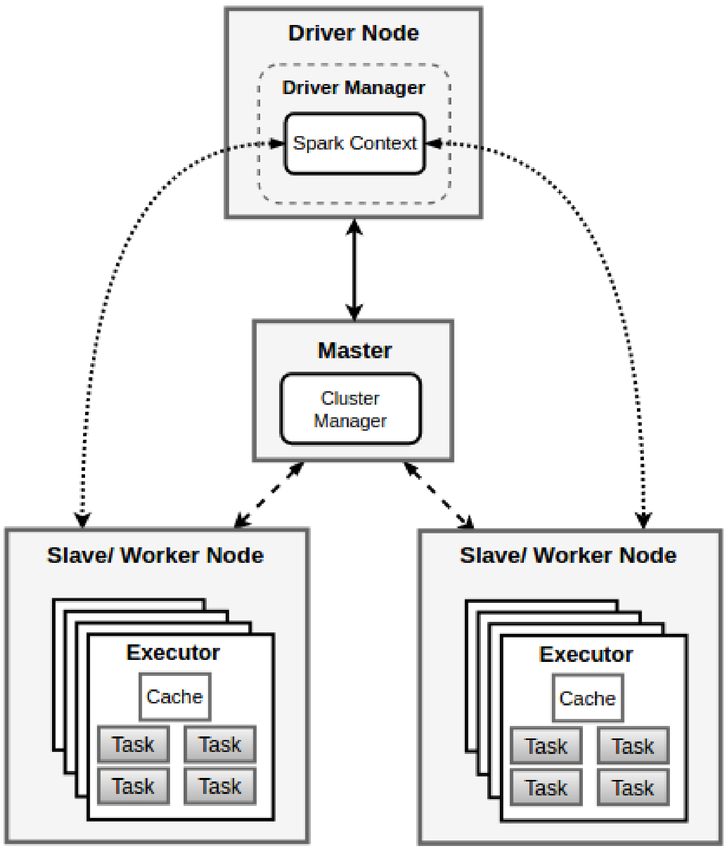


Figure 2.10: Spark cluster architecture.

2.3.3 Scheduler

As described before, Spark application has a SparkContext that runs a group of executors. One Spark application has multiple jobs that run in parallel across Spark cluster. That is why in Spark we have scheduling across different applications and another scheduling within one application.

- **Scheduling across applications:** In case of running many Spark applications in the same cluster, resources are assigned statically for each Spark application so that each application takes as much resources as it needs for all its execution running time. For Standalone cluster, applications are running in FIFO order taking all the available resources. To bound resource assignment, a max values for allocated CPU cores and memory are set in Spark configuration. Using other resource management framework such as YARN and Mesos [36], static scheduling across applications can be applied. A special dynamic scheduling is applied when using Mesos. This dynamic scheduling enables resources sharing when any Spark application is no longer using its assigned resources.

- **Scheduling within applications:** Within one Spark application, the default jobs scheduling is FIFO. This scheduling technique becomes unfair when the first job in the queue is large and resources hungry because it will delay the other applications until it finishes. Another scheduling technique is Fair scheduler which assigns resources/tasks to applications job in round robin scheme. Fair scheduler introduced the idea of Fair Scheduler Pools, where each pool is assigned different weight or resources allocation priority [37].

2.3.4 Limitations

Although Apache Spark is general and fast big-data processing tool, it has some crucial drawbacks. Here we will mention the main disadvantage of Spark that eliminate it from being a good choice for this thesis work [38, 39]:

- In the use cases that require low latency such as stream processing, Apache Spark is not the good option to cope with low latency requirements.
- Spark is fast because it utilizes memory resources and keeps processing operations always in-memory. So, Spark needs large memory requirement.
- In-memory processing in an expensive processing paradigm which is not cost-efficient in case of large clusters.
- Spark does not fully support real time processing.
- To get the better out of Spark, manual optimization is needed.
- Important feature like Back Pressure Handling is not done automatically at Spark. User has to manually adjust Spark to overcome the cases when input-output queues are full and no space to receive new data.

2.4 Apache Storm

2.4.1 Overview

Storm is an open source distributed framework that processes batch or real time stream processing in a fast and scalable way. Big-data benchmarking tools claim that Storm can process up to millions of data tuples per second [40]. Generally Storm is known for its ease of use, fault-tolerance capabilities, scalability, fast and reliable processing. The fault-tolerance is in the sense of monitoring all the cluster nodes and in case of any node failure; it can be restarted seamlessly and automatically. The scalability is shown in Storm elastic throughput that can reach up to one million bytes per second and the parallel execution of complex operations across Storm cluster. Finally, Storm is designed to be easy configured and deployed with standard configurations provided [41].

2.4.2 Components and Cluster Architecture

Storm main logic components are Spouts and Bolts which construct a Storm Topology. A topology is the network that represents the logical connections between group of spouts and multiple layers of connected bolts as shown in Figure 2.11.

A Storm topology is analogues to a directed graph where edges are the data flow between nodes and vertices are the computation in spouts and bolts nodes [42]. Spouts usually receive streams of data from external data sources such as Kafka [41], Kestrel [42] or Network File Systems (NFS) [41] then they feed these data streams into Storm Bolts. Bolts are the processing engine of Storm where the entire data transformation and processing logic take place such as aggregation, filters, joints or user developed functions. Bolts can have multiple input or output streams at the same time sent from Spouts or other Bolts [41, 43]. Data are processed within Storm as streams which are a continuous sequence of data tuples in the form of key-value pairs. Tuples can contain various types of data such as the regular data types: integers, longs, Boolean ...etc. or you can define your custom data type using serializers.

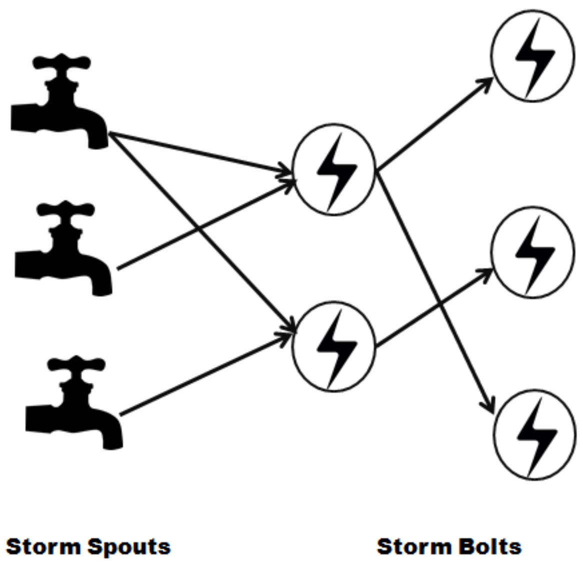


Figure 2.11: Representation of Storm topology containing spouts and bolts.

In order to achieve higher processing parallelism, Storm runs one virtual machine (VM) per one physical slave called *supervisor*. Each supervisor can run one or more Java Virtual Machines (JVMs) called *workers* that belong to one or multiple running Storm topologies. Each Strom *worker* has multiple threads *executors* that may be a Spout or a Processing Bolt. To increase the parallelism level, one executor can run multiple entities *tasks* of a specific spout or bolt. Tasks are the actual processing units that are distributed across Storm cluster machines [44]. Number of Storm supervisors equals to number of physical machines in Storm cluster, while number of workers is configured in Storm configuration file *storm.yaml*.

Storm cluster architecture consists of group of physical machines: the master and worker nodes. The master node runs a Java daemon called *nimbus* which is responsible for distributing the processing tasks over the worker nodes. Storm topology is first submitted to Storm nimbus which distributes input data between Storm spouts and monitors the execution of the topology within worker nodes. So, nimbus can assign tasks to worker machines and detect any node failure. Nimbus is a Thrift service where you can submit your topology in any preferred programming language and Storm will understand it [44], [45].

On the other hand, worker nodes run a daemon called *supervisor*. Each worker node has a group of worker processes. Supervisor distributes the workload over these worker processes such that each worker process belongs to one running topology [46]. Each worker process executes a subset of a topology function. Note that every worker process has multiple threads called executors that hold one or more tasks where the actual logic of spouts or bolts is done. When a topology is submitted to Storm nimbus, the entire topology code and its dependencies are packaged into a JAR file that is distributed over the cluster worker nodes by Storm nimbus. The communication between nimbus and supervisors is governed by an entity called a *zookeeper*. Zookeeper holds the state of nimbus and supervisors in local disk so in case of node failure; nodes can restart from where it crashed [43]. Storm cluster can have one or more zookeeper entities. The abstract architecture of Storm cluster is shown in Figure 2.12 also Figure 2.13 depicts the internals of one Storm worker node.

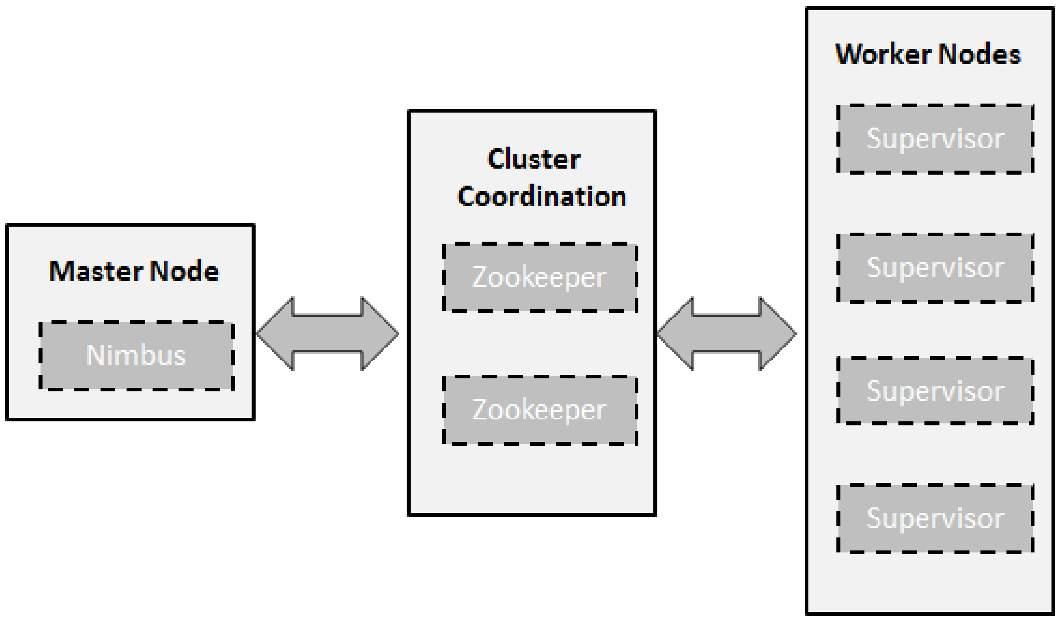


Figure 2.12: Storm abstract architecture.

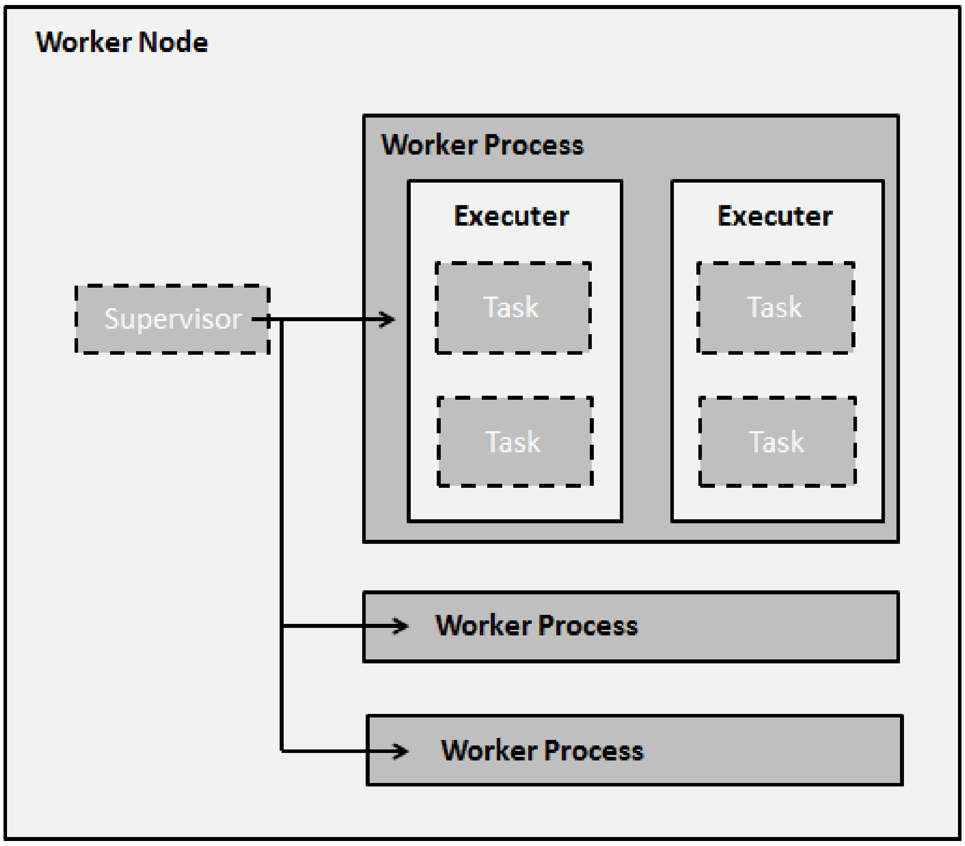


Figure 2.13: Storm worker node internals.

2.4.3 Scheduler

In order to run Storm topology probably and utilize the available resource efficiently, Storm needs a scheduler that allocates all the available workers to cluster worker nodes. Storm Default Scheduler is called *Even Scheduler*. Its strategy is to allocate the entire resources evenly between worker nodes using random round-robin policy. Scheduling here is performed in two steps: the first step is to allocate the topology executors to workers in round-robin fashion. The second step is to assign these workers to worker nodes in even distribution, so that all worker nodes are assigned nearly the same number of workers [45].

Although the default scheduler attains fairness among worker nodes, it has some limitations and drawbacks. From the disadvantages of *Storm Even Scheduler* is that users cannot know the exact distribution of their topology components over the cluster, however it may be needed to assign specific tasks to certain topology components [47]. Also resource allocation using default scheduler may not be the optimum allocation that satisfies all the tasks requirements [48]. Another shortage of the default scheduler is that it is not a dynamic scheduling that cannot cope with the ongoing increasing demands of a running topology. Finally, the default scheduler does not consider many vital aspects that affect Storm performance such as: the communication overhead between nodes, streams transmission time, resources specification and tasks requirements.

In order to satisfy the user's specific scheduling policy, Storm gives the ability to design custom schedulers using an API called *IScheduler*. To plug-in new schedulers, we need to modify the IScheduler interface and feed it with all the necessary information about the running topologies and deployed cluster in addition to the users special requirements [45]. Within Storm supervisors configuration, there is a special field called *storm.scheduler.meta* which identifies the used custom scheduler in a key-value pair [47].

Examples of Storm Custom Schedulers are Adaptive online Scheduler [45], Resource Aware Scheduler (R-Storm) [49], Metadata Aware Storm Scheduler [50] and QoS-aware scheduler [51].

2.4.4 Storm Stream Grouping

As illustrated, Storm topology has a graph of connected spouts and layers of processing bolts where each bolt executes many tasks such that each task gets subset of the data tuples sent from spouts. The partitioning of data streams down to processing bolt tasks is governed by some built-in stream grouping types [48]. The available stream grouping types in Storm are [48, 52]:

- **Shuffle Grouping:** When there is no exact requirements for distributing data tuples, shuffle grouping is used. Basically shuffle grouping partitions tuples randomly and equally between bolt tasks to ensure uniform load balance and equal number of tuples are sent to each task.
- **Fields Grouping:** In this grouping type, data tuples with same specific field value will go to the same bolt task for processing. Tuples with another field value may or may not go to the same bolt task, here we only ensure that data with specific field value will be grouped together in same processing unit.
- **All Grouping:** This grouping does not partition data tuples, but it sends a replica of all tuples to each bolt task in the topology.
- **Global Grouping:** It is a special case of All grouping, however here all the tuples belong to one stream are sent to one bolt task which has the lowest ID.

- **Direct Grouping:** Here the tuple emitter (usually Spout) decides where exactly each tuple will go for processing in the topology components. This requires data streams to be declared as Direct streams when configuring the topology to be able to direct these streams to specific bolt tasks.
- **Local or Shuffle Grouping:** In case of having many tasks for the same bolt, this local grouping shuffles tuples between these bolt tasks. Otherwise, it will act as the regular shuffle grouping.
- **None Grouping:** It is used when there is no exact preference when partitioning data down to bolt tasks. Commonly None grouping acts as shuffle grouping.
- **Custom Grouping:** Storm gives an API to implement a custom grouping that fulfill your needs when all the built-in grouping techniques cannot fit your grouping requirements.

Figure 2.14 shows a graphical representation of some of the stream grouping types used in Storm.

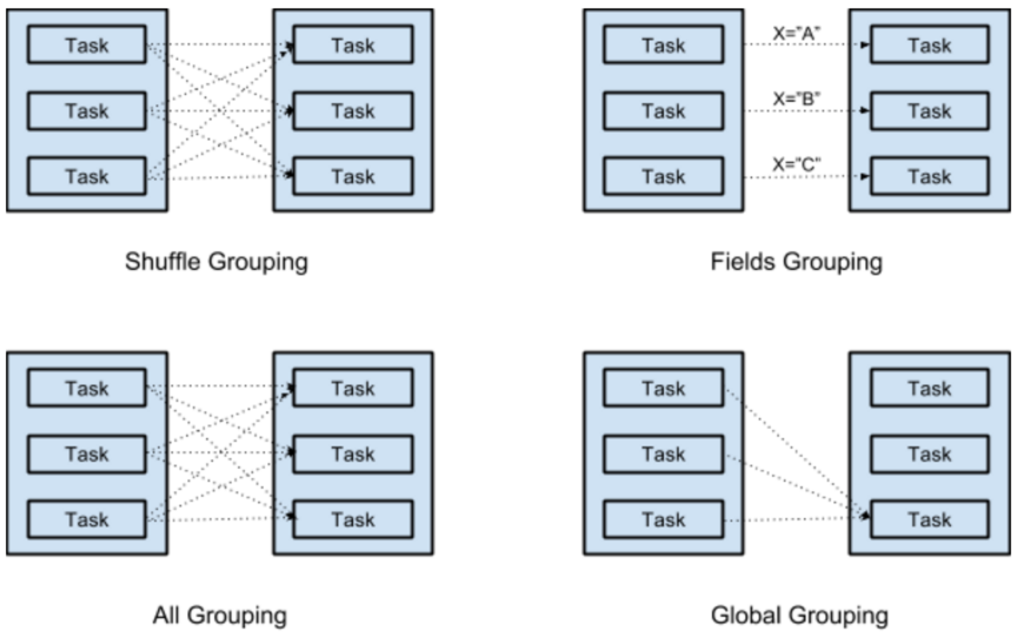
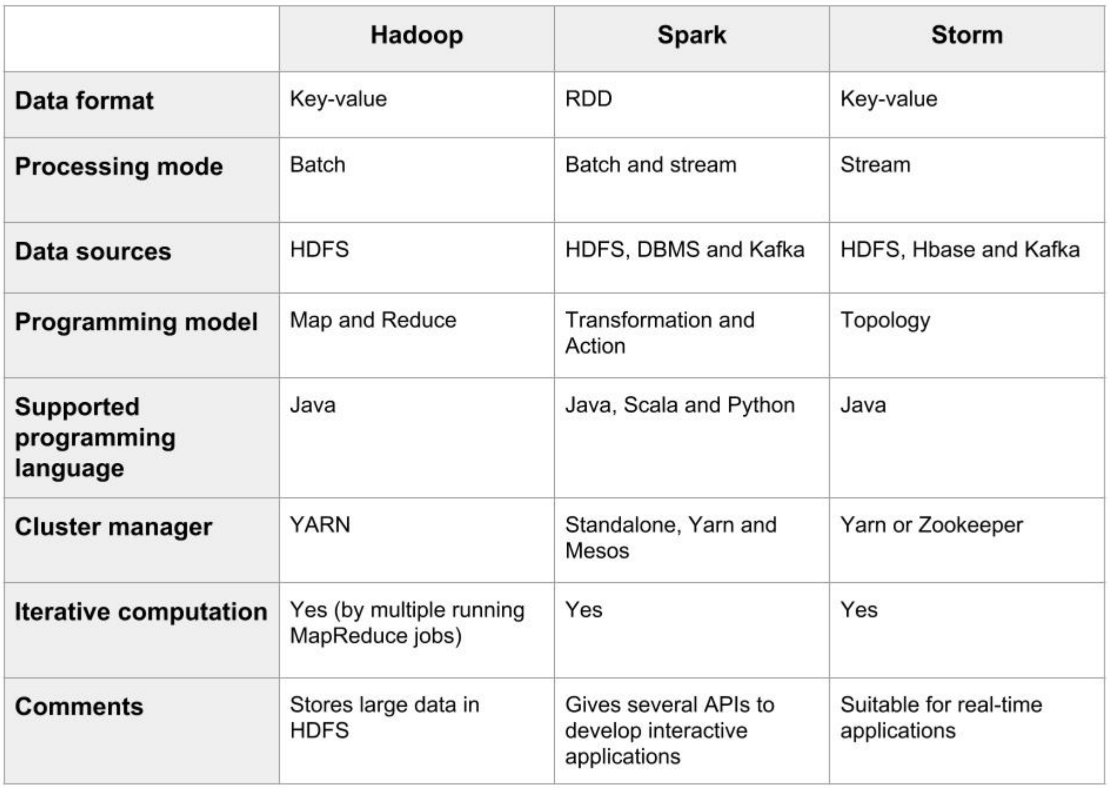


Figure 2.14: Different stream grouping types in Storm [52].

2.5 Comparisons between Big-Data Technologies

To summarize the main features of Hadoop, Spark and Storm, a quick comparisons is conducted in Table 2.1 which focuses on the key differentiating points between these big-data technologies [17]. This comparison shows how powerful Hadoop, Spark and Storm are in processing and handling big-data and how each one of them was designed and tailored to overcome its predecessors drawbacks. From this comparison and the aforementioned limitations of both Hadoop and Spark, Apache Storm is found to be the best big-data platform for the research in this thesis.

Table 2.1: Comparison between big-data technologies [17].



	Hadoop	Spark	Storm
Data format	Key-value	RDD	Key-value
Processing mode	Batch	Batch and stream	Stream
Data sources	HDFS	HDFS, DBMS and Kafka	HDFS, Hbase and Kafka
Programming model	Map and Reduce	Transformation and Action	Topology
Supported programming language	Java	Java, Scala and Python	Java
Cluster manager	YARN	Standalone, Yarn and Mesos	Yarn or Zookeeper
Iterative computation	Yes (by multiple running MapReduce jobs)	Yes	Yes
Comments	Stores large data in HDFS	Gives several APIs to develop interactive applications	Suitable for real-time applications

When designing video processing and computer vision framework, we need a scalable, fault tolerant and real time tool such as Storm. Storm is more flexible in implementing special processing logic unlike Hadoop and Spark which are limited by MapReduce or DAG framework. This flexibility is required for computer vision algorithms that may include several processing or data reducing stages that can be achieved using multi-layer Storm topology. Also Storm has no single point of failure. If nimbus node dies, the running job will remain functional and nimbus itself will restart easily. If supervisor node dies, all jobs waiting in the queue will be reassigned to new supervisor nodes without data loss. These fault-tolerance capabilities are crucial for video processing applications that are related to security and governmental sectors. An important aspect to choose Storm is the stream processing which enables video processing in real-time. Apache Storm currently has a wide adoption by well-known technological companies such as YAHOO, Twitter, Spotify and much more [40].

2.6 Conclusion

In this chapter we had a comprehensive illustration about three of powerful big-data tools; Hadoop, Spark and Storm. We presented an overview about each tool development history and how it evolved to its current state as a powerful and robust processing tool. We talked about the main features of each tool and the its key competencies. A brief discussion about the processing framework components and cluster architecture for Hadoop, Spark and Storm was introduced to understand how each tool works and how to tune anyone of them for specific use cases. We found that Hadoop and Spark have various limitations that hinder them from being good candidate for the developed computer vision processing framework in this thesis. Apache Storm was shown as the best big-data framework technology that suits this thesis objective specially for computer vision and video processing requirements.

Chapter 3: Literature Review

3.1 Introduction

In this chapter, we will go through the basic concepts of computer vision and its objective. We will highlight some previous efforts in enhancing sequential computer vision algorithms and performing parallel processing using big-data technologies and how these efforts differ from this thesis objectives.

3.2 Computer Vision and Video Processing Overview

Computer vision (CV) is known to be the field of developing algorithms that enable computers and machines to gain knowledge from digital images or videos. CV field aims to mimic the perception of the human to the surroundings and how the human visual system understands and analyzes the context from images or videos [53]. Generally, CV algorithms target the processing of different media forms such as images, sequence of videos or shots from multiple cameras to automatically extract useful information about the content of these media sources. Such important theories and algorithms are used in various applications and fields.

CV goals span from detecting an object appears in image or sequence of video, specifying the object position, understanding the actions being performed in the scene, getting the meaning behind theses actions till taking decisions based on the conclusions acquired from the captured details [54]. The general steps to acquire information from digital image or one video frame are shown at Figure 3.1.

First, pre-processing algorithms such as noise reduction and image scaling are required to alter and fine tune the image under processing to match the processing algorithm requirements. The second step is to select the Area/Region Of Interest (ROI) for the algorithm which means that we only want to apply the processing over specific segment of the provided image and not the entire frame. This can be done using different techniques such as Image Segmentation, Object Detection and Background Subtraction. Then the selected area is given to the processing algorithm that has a specific logic to apply in order to perform operations such as Object Detection, Tracking and Feature Matching. The final step is to reach precious decisions based on the algorithm results and extracted information from each individual frame. CV final results may include Motion Analysis, Pattern Recognition, Alerts for specific events or Summaries of video content and much more [55].

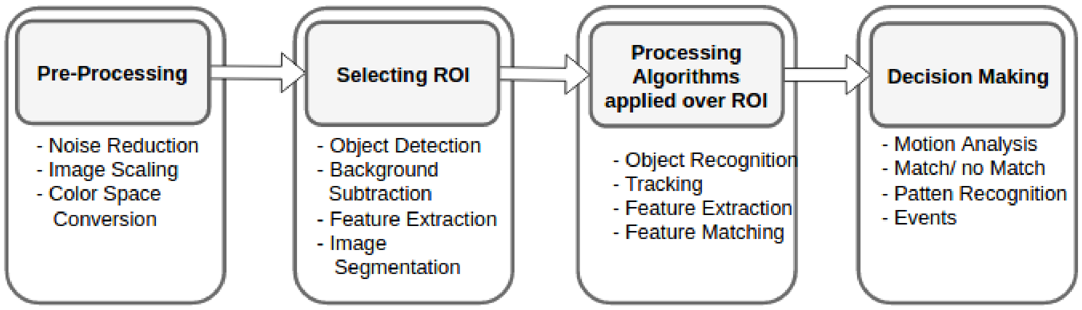


Figure 3.1: Computer vision steps sequence [55].

CV and Video Processing (VP) have various applications in diverse fields. Examples of CV applications:

- Optical Characters Recognition (OCR).
- Face Detection.
- Scene Reconstruction.
- Object Tracking.
- People Count.
- Smile Recognition.
- Object Recognition.
- 3D Modeling.
- Video Summarization.
- Indexing of image/video databases.
- Event Detection.
- Safety Alerts.
- Traffic Monitoring.

3.3 Previous Work

Enabling video processing and analytics in big-data frameworks gave the advantage of parallelizing such processing algorithms and getting the maximum out of the underlying hardware resources. To parallelize CV algorithms, many previous contributions were done in several directions regarding enhancing the processing hardware itself, changing the algorithm software structure or utilizing the current sequential hardware in a parallelized way. The earlier efforts went on the direction of parallelizing the hardware architecture whether it is CPU or GPU architecture.

As mentioned in [56] the clock frequency of CPUs reached almost a fixed value in 2004. So, in order to increase the parallelism of CV algorithms using such limited sequential hardware, we have to utilize the available commodity multi-CPU multi-GPU hardware. The work done by Jaap Van targeted the acceleration of sequential CV algorithms using a conventional PC that has a multi-core processor in addition to one or more attached GPUs. The author used OpenMP [57] to benchmark the CV algorithm performance over parallel multi-core CPU. Using OpenMp, Van achieved speed up from 2.7 to 5 when processing large size images on a quad-core Intel i7 running Windows 7 using sequential algorithms while a processing overhead was added when processing small images.

CV processing is categorized into 3 levels[58]; *Low Level* which includes pixel-based transformations, *Intermediate Level* which is related to selecting region of interests from processed images and apply numeric or symbolic operations and *High Level* which combines low and intermediate level results and concludes to information related to object recognition. The work proposed in [59] was concerned with parallelizating low-level CV algorithms on distributed cluster of workstations. Two cluster patterns were introduced; Farmer-Worker pattern and Master-Worker pattern. To process independant segments of images data, the farmer-worker paradigm is used where data is parallelized and processed in independent workers. Unlike farmer-worker, the master-worker architecture has a master node that distributes data processing among workers while guarantees communication and data sharing between those workers. A significant speedup was obtained using farmer-worker architecture in applying images convolution operation beside speedup achieved at image restoration algorithm using master-worker architecture.

Video Transcoding is the process of converting the representation of video data from one format to another. Video transcoding has many objectives such as rate reduction or resolution reduction in the scope of redundancy or irrelevancy mitigation. Such intensive processing video algorithm requires high computational resources. Many efforts were done to parallelize video transcoding operations that run sequentially over distributed architecture. In [60], the author introduced the idea of parallelizing video transcoding operations using distributed computing to benefit from the scalability and availability that are offered by distributed architecture. Figure 3.2 shows how distributed computing can achieve higher throughput by adding new working nodes to the system. Also processing scheduling algorithm adopted by distributed computing can produce some performance gain. The distributed video transcoding architecture implemented in [60] is built over cloud computing using master-workers paradigm. The input video data is split into smaller parts where each part is considered an independent Group of Pictures (GOPs) -will be illustrated later in chapter 4-, see Figure 3.3 for the entire implementation. As the video split size decreases, the proposed system shows higher performance in terms of significant reduction in startup time and total transcoding time.

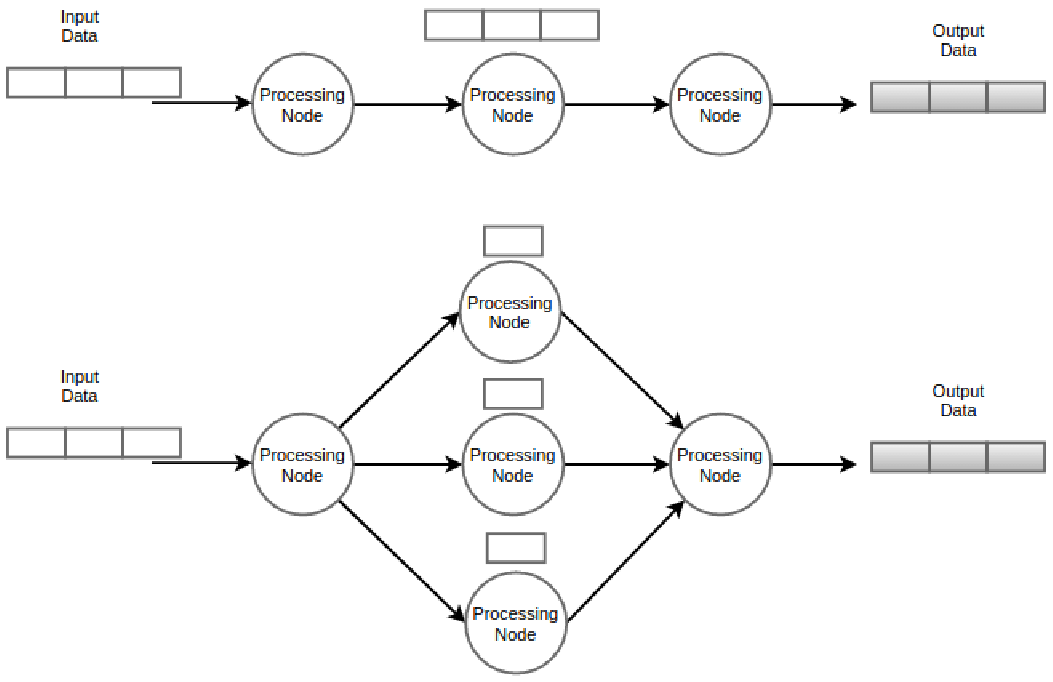


Figure 3.2: Distributed stream processing [60].

Another implementation to parallelize video transcoding on cloud computing is done using Map-Reduce-Based processing framework in [61]. The input video here is divided into small segments based on GOPs where each segment is directed into one map function and later the results out of all maps are aggregated at one reduce entity as depicted in Figure 3.4. The work done by authors targets the scheduling of sub-tasks to cloud computing machines in order to achieve better performance. The scheduling follows Max Minimal Complete Time (Max-MCT) approach in assigning video split processing task to working machines. The segment length, computer capacity and task-launching overhead are the considered parameters in evaluating the developed scheduling algorithm. The Max-MCT tasks scheduling outperformed original MCT however it is not the most optimized scheduling technique.

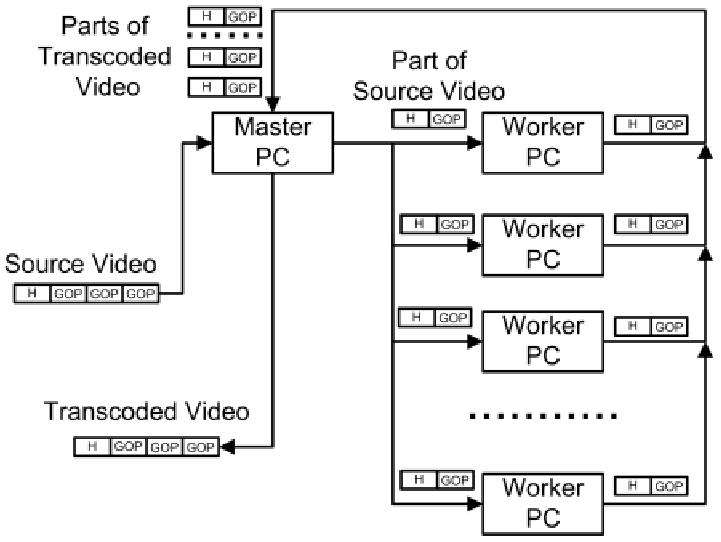


Figure 3.3: Distributed video transcoding abstract architecture [60].

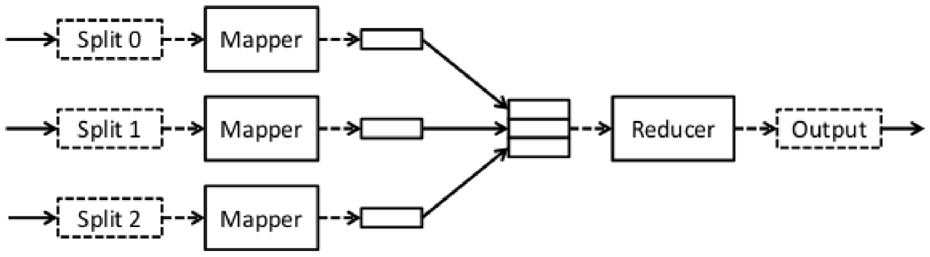


Figure 3.4: Video transcoding using mapreduce-based cloud computing [61].

A number of big-data analytics platforms were built using Hadoop framework. In [62], Pivotal [63] Data Science team established a project that implements large-scale video analytics over Hadoop. The analytics platform aims to process incoming video streams in real time, extract important insights and trigger alarms or take actions based on the analytics results. Video transcoding is done in parallel within HDFS over video chunks using MPEG-2 encoding/decoding standards where videos are transformed into a Hadoop sequence file of image frames for better handling. Video analytics in this system is governed by MapReduce processing framework that suits a diverse collection of computer vision algorithms such as feature extraction and object detection. To extract structured data out of the unstructured processed video streams, SQL, the advanced query language, was used. The performance of this system was evaluated using Hadoop six-nodes cluster where the video transcoding step and video analytics is performed in minutes over gigabytes of long-hours videos.

Authors work in [64] targets the design of video monitoring system that is concerned with big-data using Hadoop. This monitoring system has vital applications to support safe city and intelligent transportation fields. The designed monitoring system focused on the challenges of big-data video monitoring regarding the required high volume data storage, the complexity of video intelligent analysis and the fusion of large data to implement useful video-based applications. The monitoring system's big-data processing platform component includes Hadoop and HDFS where actual CV and data analytics are done. In order to provide the user with good video browsing service, a video distribution network was added to this system to overcome the weak random read of HDFS. Another customized components such as Task Scheduling and Management and Application Service Interface were added to enhance the overall big-data video monitoring system and provide high quality services in real time as depicted in Figure 3.5.

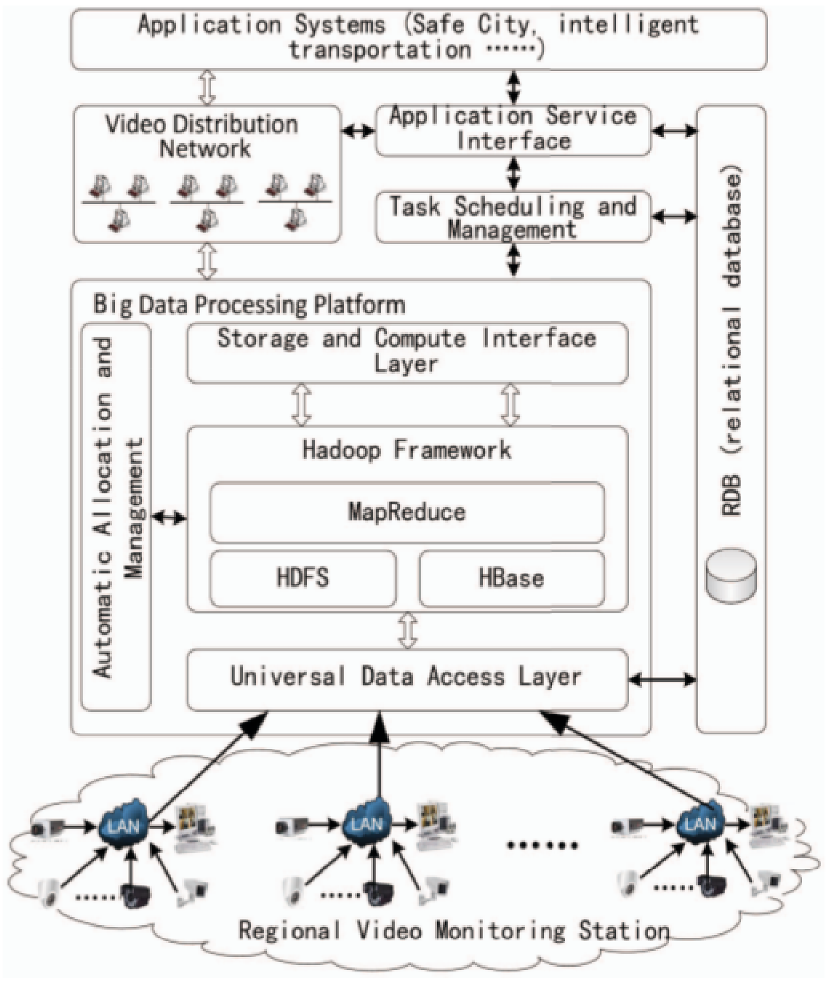

Application Systems (Safe City, intelligent transportation ……)
Video Distribution Network
Application Service Interface
Task Scheduling and Management
RDB (relational database)
Big Data Processing Platform
Automatic Allocation and Management
Storage and Compute Interface Layer
Hadoop Framework
MapReduce
HDFS
HBase
Universal Data Access Layer
LAN
……
Regional Video Monitoring Station

Figure 3.5: Big-data video monitoring system based on Hadoop [64].

The work in [65] uses Apache Storm to implement a low-delay video transcoding in distributed computing platforms. The typical initial Storm topology for the proposed system is shown in Figure 3.6 where the spout is responsible for reading GOP video segments and providing them to transcode bolt which performs the actual video transcoding process. The transcoding step is time consuming, that is why it is better to increase the parallelism of transcode bolt by using more workers. The transcode bolt emits transcoded video segments to MP4Mux bolt which is an optional step based on user configuration. Finally the dash bolt collects all video segments. For multi-camera option, the author stated that the topology components shown in Figure 3.6 should be replicated to handle different video data generated from multiple camera sources. The author implemented a custom Storm scheduler to force spouts and dash bolts to be scheduled to same Storm supervisor. The overall system was tested for local and cloud computers using 720p, 1080p and 4K videos. The 720p video segments gave the best results.

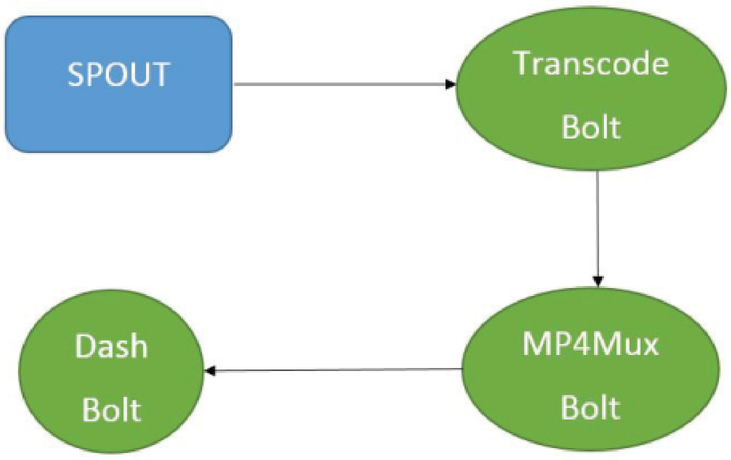


Figure 3.6: Low-delay Video Transcoding initial topology [65].

To utilize the distributed processing capabilities in CV operations, a new platform called StormCV [66] was designed to enable Storm to perform CV functions within the topology architecture. StormCV comes with its CV operations and data models such as frame and feature extraction. StormCV uses OpenCV to implement most of CV functions such as face detection and feature extraction. The input data types that could be processed using StormCV are video streams, video files and images. Using StormCV, the authors in [67] designed a scalable framework for video surveillance. In this system, the topology spout acts as *Stream-Frame-Fetcher*. It reads frames from video stream using Real Time Streaming Protocol (RTSP) and emits them as a *Frame Object*. A *Background Subtraction Bolt* receives frames from spout then emits them to the processing bolts with *Face Detection* or *Person Detection* functions based on the required output. To gather meaningful output from this surveillance system, the two processing bolts emit the data to a *Lapeller* then *Export-to-File Bolt*. A standalone *Export-To-Video Bolt* is running to save summarized output videos if needed. A complete realization of Storm topology using StormCV as per authors design is shown in Figure 3.7. To evaluate the proposed surveillance system, the authors compared the reduction in output videos size using a large person dataset which resulted in almost 85.65 percent decrease in disk usage based on the video content.

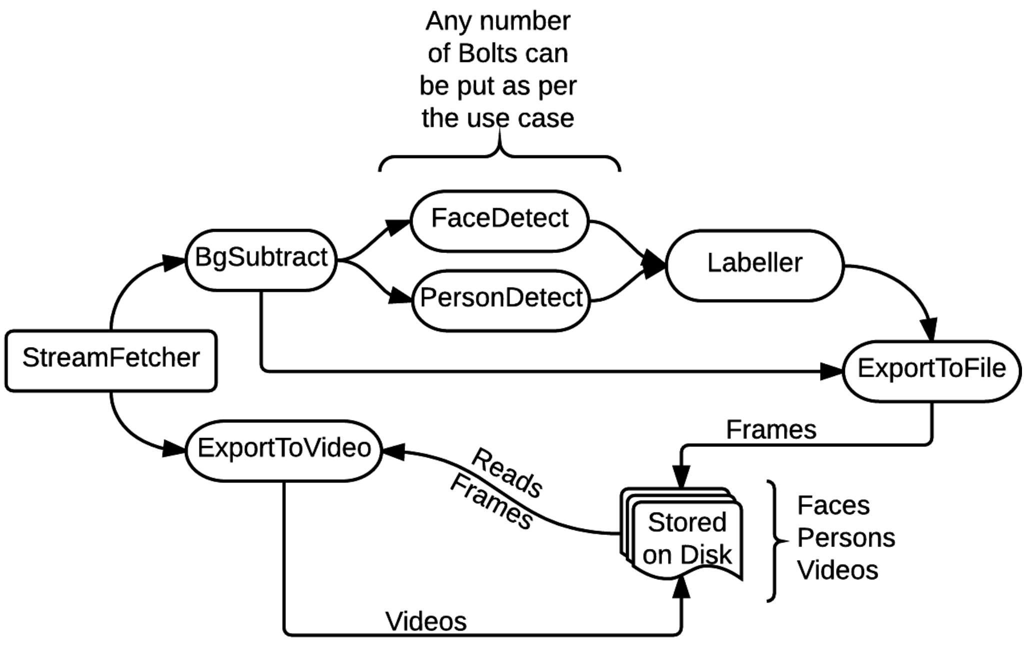


Figure 3.7: Storm topology for scalable and intelligent real-time video surveillance framework [67].

3.4 Conclusion

In this chapter we discussed the concept of CV and the steps followed to acquire information and insights from images and video data. We reviewed a collection of previous efforts done to parallelize the execution of CV algorithms such as video transcoding using multi-core distributed commodity hardware and within cloud computing architecture. Many developed systems tackled the idea of enabling big-data processing frameworks such as Hadoop and Storm to support CV functions and requirements. Although all the discussed work gave significant improvements in CV parallelization, they have some notable limitations:

- *Not Generic Systems:* Most of the designed systems target the scaling of specific CV algorithms such as video transcoding and surveillance-related algorithms and not generic.
- *Deep knowledge of CV is required:* In order to implement any CV algorithm using any of the aforementioned systems, the user needs to have deep knowledge of particular libraries and coding functions in order to do any modifications in the used algorithms.

- *Not suitable for sequential CV algorithms:* The direct enhancement of these efforts goes for CV algorithms that can run already in parallel with no inter-frame data dependency, however sequential CV algorithms with inter-frame dependency have vital applications in most fields.
- *Enhancement evaluation is algorithm-specific:* To assess the advancement of these previous efforts, the authors focused on observing evaluation metrics that are related to the algorithms under study such as reduction in output video size which may not be applicable to other CV algorithms.

In the next chapter we propose a generic scalable distributed chunk-based video processing framework that targets using Apache Storm for big-data sequential CV algorithms.

Chapter 4: Distributed Chunk-based Big-Data Processing Framework

4.1 Introduction

In this chapter a new processing framework is introduced to process sequential CV algorithms in parallel using big-data tools. We will illustrate the framework architecture and all the new developed components that cover the entire VP pipeline from external data ingestion to completing data processing along with taking the decisions needed for efficient resources usages using Apache Storm. First, we will illustrate the basic concept of the introduced chunk-based framework and highlight its new features. Second, we will go through the new system architecture and the novel components with a detailed description of each component functions and responsibilities. We will talk about all the challenges of sequential VP mentioned in chapter 1 and how our new chunk-based processing framework overcomes them.

4.2 Basic Concept

CV and VP algorithms incorporate several stages to read images or video frames, analyze the data within each input sample and extract useful results and insights based on the processing algorithm function as previously illustrated in Figure 3.1. For each individual video frame processing, the data extracted from this frame may have a direct dependency on the previous frames in sequence or it is self dependant. In case of no dependency between the processed frames, we can process all the frames in parallel with random order using big-data tools such as Apache Storm. Although the frames are dependant, we may have a redundancy in the output data due to replicated objects and events occured within the adjacent frames. Using mapreduce framework eliminates the redundancy in the results by using a reduce function that combines the output of each frame and concludes to clear and robust results.

When it comes to sequential VP where an inter-frames dependency exists, it is needed to process video frames in order of their presence at original video file. The results of each video frame are used in processing the next coming frame. When feeding video frames to big-data tools one by one especially Storm, they cannot cope with the dependency nature of such sequential algorithms. Storm will distribute these frames randomly to the available processing bolts without tracking their order or paying attention to the sequential processing requirements. In order to overcome Storm limitation in running sequential VP algorithm besides benefiting from Storm powerful big-data capabilities, we developed the distributed chunk-based big-data processing framework (DCB-Framework). The DCB-Framework is built using mapreduce framework and customized for Apache Storm.

The basic idea of the DCB-Framework is to take input video files and divide each file into small video chunks to be given to Storm topology as independent inputs. Each video chunk will be processed by Storm bolts frame by frame in order without the need to go through the entire video file in sequence. The chunk size value used to split each input video along with total number of chunks is calculated based on the available CPU cores and minimum number of frames per chunk that produces meaningful output. The value of minimum chunk size can vary from one VP algorithm and another. For example, algorithm such as video summarization needs minimum number of frames lets say 300 in sequence frames to perform background subtraction and extract objects. So, it is not feasible to pick a chunk size smaller than 300 frames to operate with the DCB-Framework for video summarization as an example.

DCB-Framework dynamically adjusts the configuration of Storm topology before it starts to satisfy the needed resources for the input video chunks. That implies the reconfiguration of the chunk size requested by the user to a more suitable value that produces number of video chunks equals to number of available CPU core. It is assumed that each video chunk needs 1 CPU core for processing. This enables us to process each video chunk in a separate core and perform the processing of all the chunks in parallel. In order to process each video chunks in parallel, we need to execute the sequential CV algorithm over each chunk in a separate processing entity that takes the chunk's frames in order and process them sequentially. A custom data grouping mechanism is developed to enable Storm grouping all the frames that belonging to specific video chunk and sending them to independent processing bolt instances. The custom grouping logic aims to maintain data consistency in each processing instance and preserve the order of processing of chunks frames in sequence.

When Storm spout receives an input data tuple which is one video frame in our case, the spout will emit the tuple to the least loaded bolt task in Storm topology for processing. However in distributed DCB-Framework it is needed to direct each frame to one distinct bolt task based on the decision of the implemented custom data grouping. In our work we added a new bolt tasks assignment technique different than the default round-robin one that Storm uses. A novel dynamic chunks distribution mechanism that maintains an assignment map between video chunks IDs and processing bolt tasks is implemented within the DCB-Framework. This new distribution mechanism monitors the distribution of all the upcoming frames one by one on the fly within Storm topology to make sure it will comply with the assignment map. This also prohibits any racing on the available resources that could occur from random video frame emitting. DCB-Framework can process multiple video files in parallel and produce the results of all files almost in the speed of processing one file individually with high accuracy.

A typical representation of a Storm topology that is used to execute CV algorithms following the DCB-Framework shown in Figure 4.1. First, the video files that needed to be processed pass through a native C++ application which reads each video file from disk and splits it into group of small chunks according to the selected chunk size. Each video chunk is treated as a sequence of video frames and sent to Storm topology using Apache Thrift [68, 69] as a communication channel. Thrift serializes each frame with its respective metadata and sends it using thrift server on specific network ports. Storm spouts listen to these specific thrift servers ports and receive the video frames and their info. Spouts act as data sources, so when they receive video frames, spouts emit the frames to Storm bolt tasks according to the data grouping criteria applied.

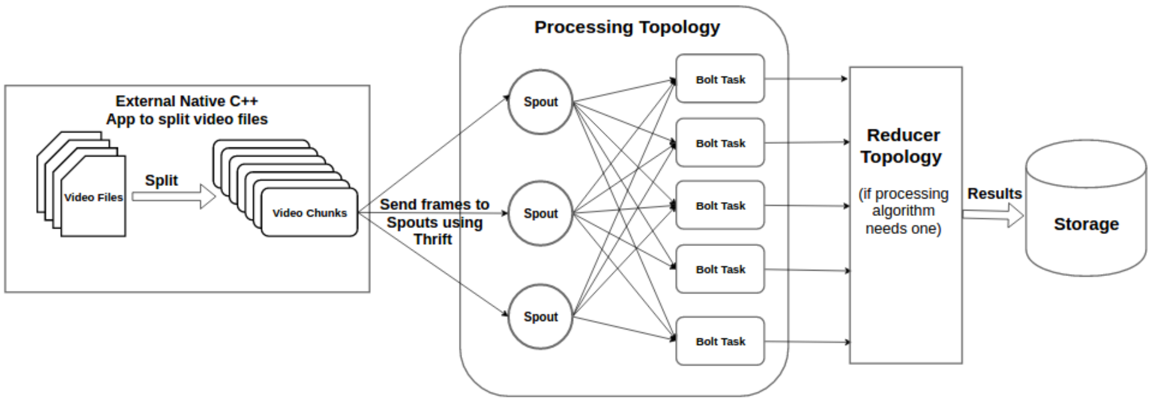


Figure 4.1: Storm topology for CV processing using DCB-Framework.

Each Storm bolt has a group of sub-tasks that contain the VP engine. Wherever a Storm topology is submitted, the entire VP algorithm libraries and dependency files are grouped in a jar file. This jar file is uploaded to working machine where bolt tasks should run. For each CV algorithm, multiple bolt task instances are initiated to parallelize the processing, increase the throughput and decrease the complete latency. Every instance executes the processing algorithm over different data received from spouts and independent of the processing that occur in other bolt instances even if all of them are located in the same cluster physical node. For bolt tasks distribution over the physical nodes, we use the default fair scheduler that Storm adopts and mentioned at Chapter 2.

In some processing algorithms, it is required to combine the results out of each video chunk to eliminate the redundancy of data produced from cutting the video file into sub chunks. For example, if the algorithm is detecting the appearance of a certain object through the entire video frames and this object just starts showing in the last frame of one video chunk and continue showing in the beginning of the next chunk. This object will be recognized as two different objects which gives misleading results. So, a reduce step is required after finishing the processing to gather such information and remove any result redundancy that happens in edges of video chunks.

As shown in Figure 4.1 the reduce function is executed in a seperate topology (*Reducer Topology*) which is submitted with the original processing topology. The spouts of the reducer topology receive the initial results of the CV algorithms through Thrift and emit them to the reducer bolts for applying reducer logic. Usually the reducer processing is not heavy compared to the processing algorithm itself. That is why the scope of this thesis experimental results is testing the distributed DCB-Framework over the processing topology itself. VP can produce different types of output data. The results could be in the form of text metadata, images or video files. When Storm bolts finish executing the CV processing logic, all the resultant data are generated and stored on persistent database or locally on disk.

From all of that we conclude that generally CV and VP algorithms can be ported within Storm big-data processing framework. Spouts are used as data sources that read input video frames and distribute them to the topology processing units which are bolts. To enable processing of sequential CV algorithms, we introduced the concept of distributed chunk-based processing in this thesis. The new DCB-Framework leverages the powerful capabilities of Storm in processing huge amount of video files in competitive time while using all the available resources efficiently.

In the next section, a detailed elaboration of the DCB-Framework is conducted. We will introduce new components to Storm topology architecture and list all their functionalities and the logic behind how they work.

4.3 Architecture

To perform the new logic of video chunks splitting and custom data grouping within Storm topology, new components and functionalities are added to Storm basic blocks. Figure 4.2 depicts an overview of the developed DCB-Framework architecture in Storm. Briefly the new working units that are presented in this thesis are :

1. **Resources Calculation Unit (RCU):** RCU evaluates the submitted topology resources requirement against the available resources. Then calculates the suitable chunk size and number of chunks according to the available resources and the number of video file(s) that needed to be processed.
2. **Data Splitting and Feed Unit (DSFU):** Here we implemented a video splitting technique called Parallel-Chunk-mode. The parallel chunk mode is used to split the video files into chunks with chunk size equals to the actual chunk size calculated in the resources calculation unit. The splitting process is done in parallel over all the video files and the chunks are sent almost in parallel to Storm spouts. This enables Storm to start processing all the video chunks almost simultaneously.
3. **Decision Making and Resources Mapping unit (DMRMU):** This unit basic function is to map all the running bolt tasks to the video chunks submitted for processing. A *Task Assignment Map* and a *Tasks IDs Queue* are created and maintained in an open source in-memory database called *Redis*. These map and queue are used to allocate the available resources and govern the distribution decisions for each video frame to assign each chunk to an individual task and avoid racing problems over the available resources. This unit also is responsible for performing data grouping where a custom direct grouping is implemented in Storm, so that each chunk is processed independently in separate Storm task.

In the upcoming sections, we will discuss all the developed components of the The DCB-Framework in more details and the relation between these components.

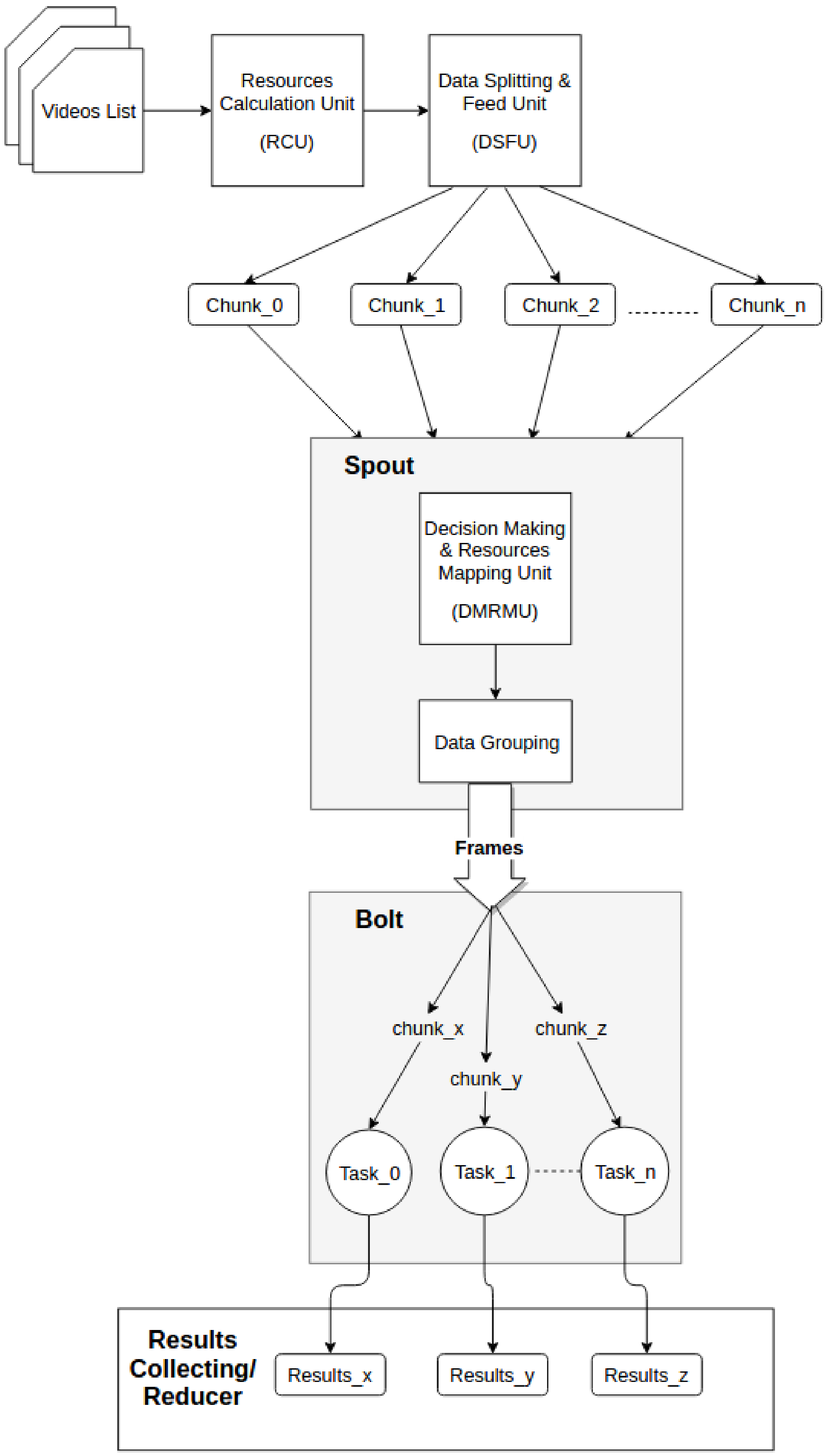

Videos List
Resources Calculation Unit (RCU)
Data Splitting & Feed Unit (DSFU)
Chunk_0
Chunk_1
Chunk_2
Chunk_n
Spout
Decision Making & Resources Mapping Unit (DMRMU)
Data Grouping
Frames
Bolt
chunk_x
chunk_y
chunk_z
Task_0
Task_1
Task_n
Results Collecting/ Reducer
Results_x
Results_y
Results_z

Figure 4.2: DCB-Framework architecture in Storm topology.

4.3.1 Resources Calculation Unit (RCU)

In order to calculate the proper chunk size and number of chunks to use at splitting video files, we developed the *Resource Calculation Unit* (RCU). The resource calculation unit takes *chunk size* as an input. The chunk size is the minimum number of frames per chunk that is preferred by the user and complies with the algorithm constraints to produce meaningful output. Some VP algorithms require certain minimum number of frames to run at once where below this number the algorithm can not operate probably.

The RCU is aware of Storm cluster overall resources, such as how many cores and memory the entire cluster machines can offer. We managed to let each active Storm topology to report its exact assigned memory and cores to Redis in-memory database -which will be discussed later-. The RCU working algorithm is as follows:

- *Inputs:*
  - List of video files to be processed.
  - Initial Chunk size: Minimum number of frames per individual video chunk that is suggested by the user.
  - Required memory per instance: This is the maximum memory that will be assigned to each bolt task instance to process an individual video chunk.
- *Functions:* Given Storm cluster actual and available resources (CPU cores and memory), The RCU is responsible for:
  - Calculation of Actual Chunk size:
  - Calculation of actual number of needed Storm bolts instances to submit Storm topology that is capable of processing the given video files in parallel.
- *Outputs:*
  - Actual Chunk size.
  - Actual number of Storm bolts needed instances.

Figure 6 shows an overview of the RCU block diagram. It depicts the inputs, basic working functions and the expected outputs of this unit.

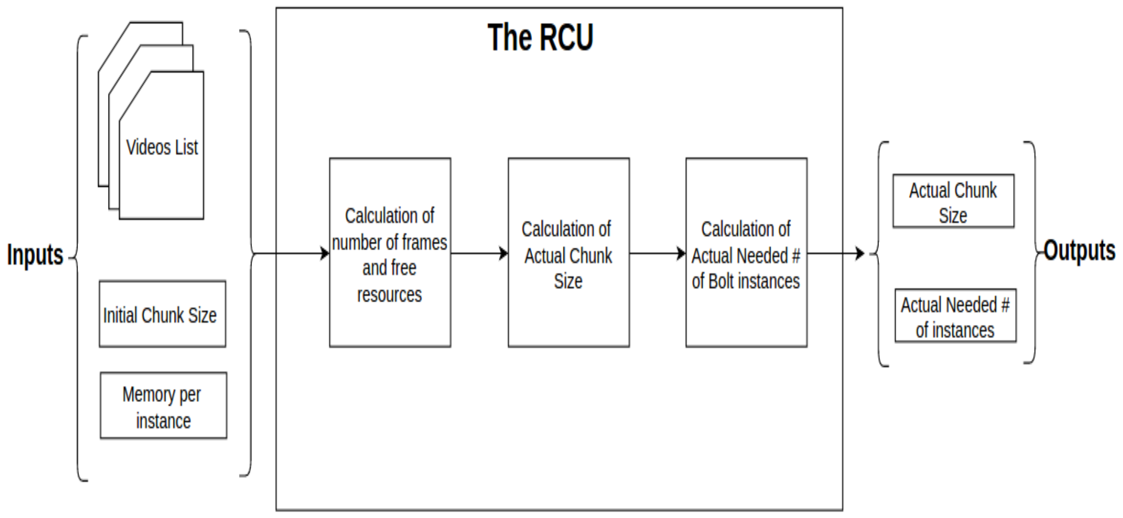

The RCU
Inputs
Videos List
Initial Chunk Size
Memory per instance
Calculation of number of frames and free resources
Calculation of Actual Chunk Size
Calculation of Actual Needed # of Bolt instances
Actual Chunk Size
Actual Needed # of instances
Outputs

Figure 4.3: The RCU block diagram.

The RCU follows this working procedure:

1. Loop over input video files list and get:
   - Number of frames per video file: this is obtained from video file specifications.
   - Total number of frames for all video files in input list (*Total # of frames*).
2. Get working cluster resources: Storm has a configuration file that describes all the cluster working nodes and their basic processing resources. Hence, from Storm cluster configuration we loop over all the cluster nodes and get:
   - Total number of available CPU cores (*Basic # of Cores*) in the cluster.
   - Total available memory (*Basic Memory*) in the cluster.
3. Get already used cluster resources: Every submitted and active topology reports its assigned cpu cores and memory to Redis. So, to calculate the used resources we do the following:
   - Get list of all topologies IDs for the running topologies from Storm. Using the topologies IDs, access Redis to get taken resources by each topology.
   - Sum the returned values to get total used cores and memory (*# of Used Cores*) and (*Used Memory*).

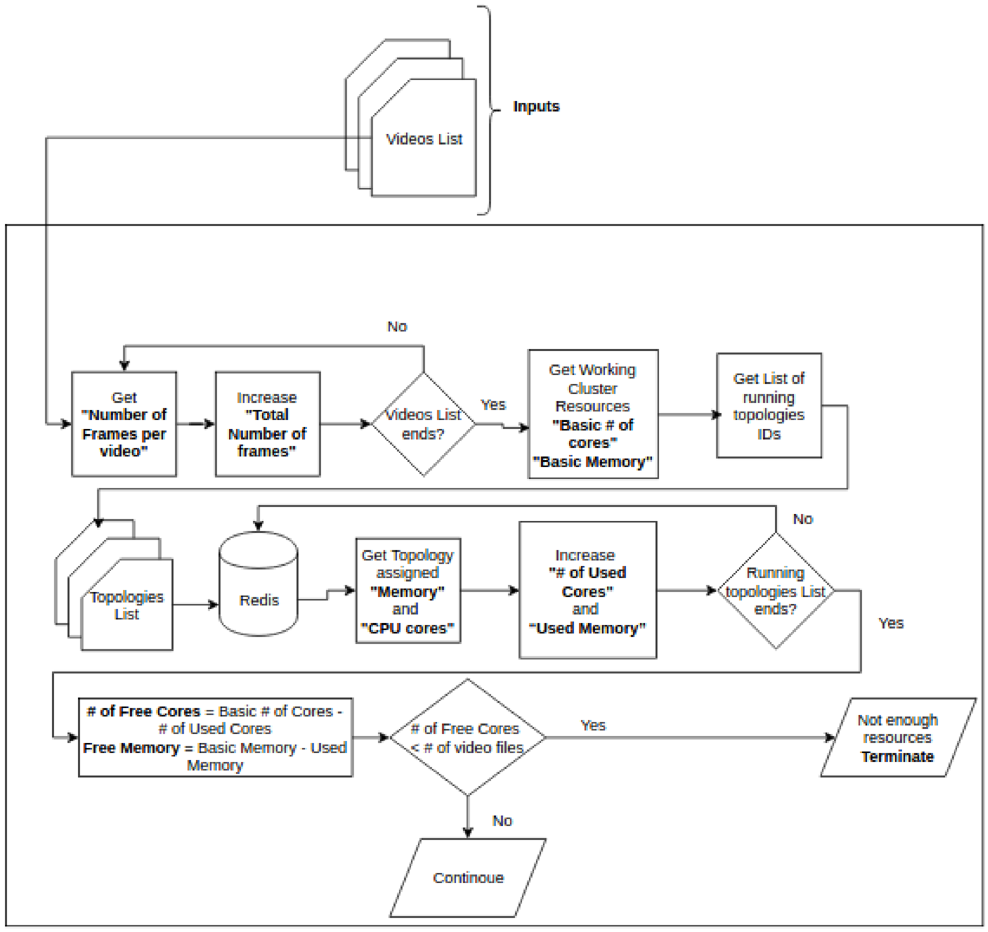
4. Calculate free resources: Having cluster basic and used resources from steps 2 and 3, we can get the free resources that are available to run a new topology in order to process the input video files list.

$$\textit{\# of Free Cores} = \textit{Basic \# of Cores} - \textit{\# of Used Cores} \tag{4.1}$$

$$\textit{Free Memory} = \textit{Basic Memory} - \textit{Used Memory} \tag{4.2}$$

If ($\textit{\# of Free Cores} < \textit{\# of video files}$) $\longrightarrow$
$\therefore$ Resources are not enough to process these files,
else continue,

Figure 4.4 illustrates the working flow of the RCU from step 1 to 4.



Videos List
Inputs
No
Get "Number of Frames per video"
Increase "Total Number of frames"
Videos List ends?
Yes
Get Working Cluster Resources "Basic # of cores" "Basic Memory"
Get List of running topologies IDs
Topologies List
Redis
Get Topology assigned "Memory" and "CPU cores"
Increase "# of Used Cores" and "Used Memory"
Running topologies List ends?
No
Yes
# of Free Cores = Basic # of Cores - # of Used Cores
Free Memory = Basic Memory - Used Memory
# of Free Cores < # of video files
Yes
Not enough resources Terminate
No
Continoue

Figure 4.4: The RCU steps to calculate number of frames and free resources.

5. Calculate number of bolt instances that can be initiated based on total free memory, required memory per instance and available number of cores in order to select the

minimum number of instances needed.

$$\textit{Initial \# of instances} = \left\lceil \frac{\textit{Free Memory}}{\textit{Memory per instance}} \right\rceil \tag{4.3}$$

If (*# of Free Cores < Initial # of instance*) $\longrightarrow$

$$\therefore \textit{Needed \# of instances} = \textit{\# of Free Cores} \tag{4.4}$$

else,

$$\textit{Needed \# of instances} = \textit{Initial \# of instances} \tag{4.5}$$

6. Calculate actual chunk size which defines number of frames per video chunk that will be used to split the video files. The calculation of chunk size is based on total number of frames of all video files in input list in addition to the calculated needed number of bolt instances. After finishing these calculations, the actual chunk size value will be the maximum between the calculated chunk size and the chunk size initial value.

$$\textit{Calculated chunk size} = \left\lceil \frac{\textit{Total \# of frames}}{\textit{Needed \# of instances}} \right\rceil \tag{4.6}$$

If (*Calculated chunk size < Initial chunk size*) $\longrightarrow$

$$\therefore \textit{Actual chunk size} = \textit{Initial chunk size} \tag{4.7}$$

else,

$$\textit{Actual chunk size} = \textit{Calculated chunk size} \tag{4.8}$$

The entire logic of calculating Actual chunk size is depicted in Figure 4.5

7. Using Actual chunk size, recalculate needed number of instance and take the minimum value between the new calculated number of instances and the one calculated in step 5.

$$\textit{New \# of instances} = \left\lceil \frac{\textit{Total \# of frames}}{\textit{Actual chunk size}} \right\rceil \tag{4.9}$$

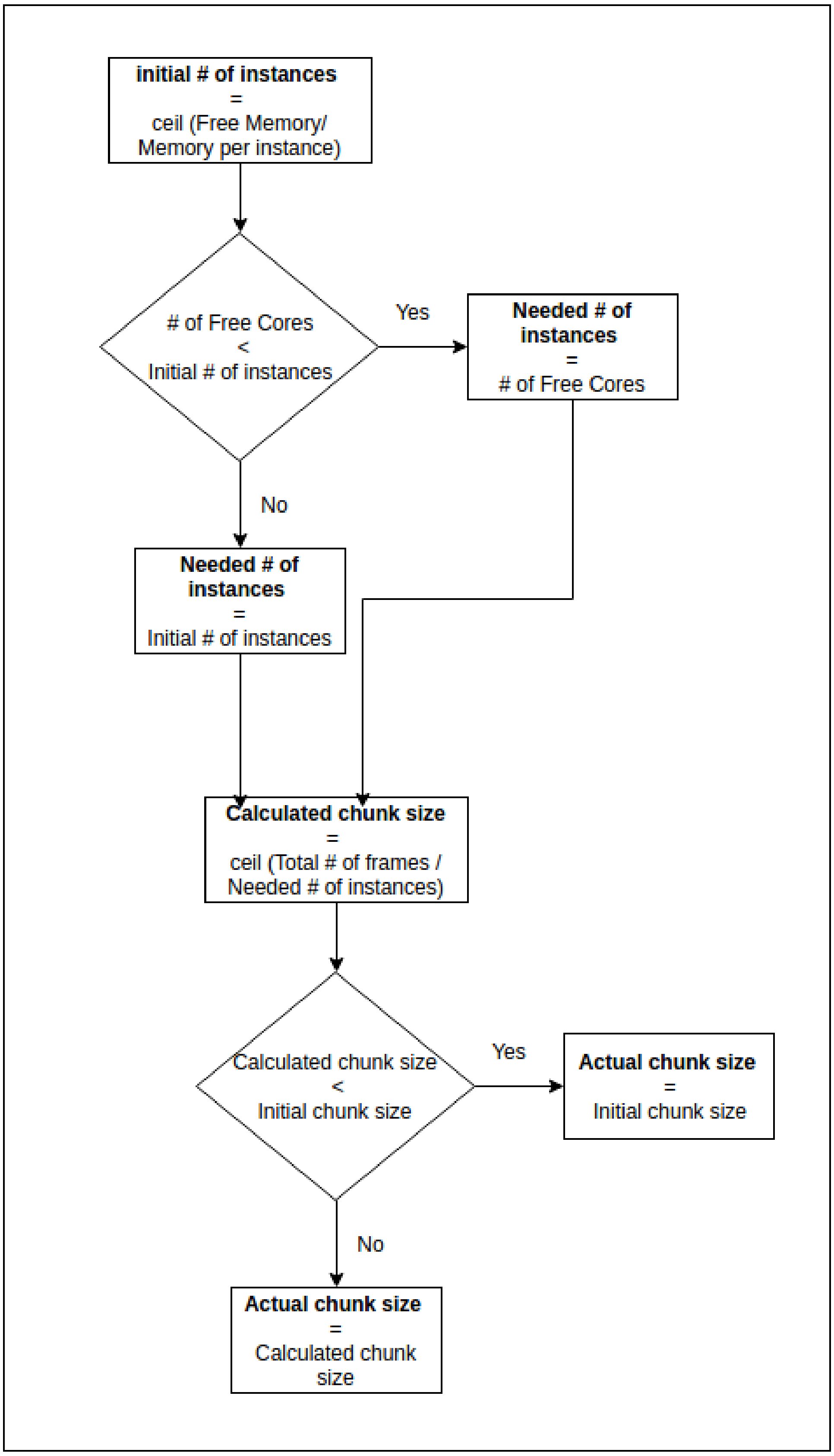

initial # of instances = ceil (Free Memory/ Memory per instance)
# of Free Cores < Initial # of instances
Yes
Needed # of instances = # of Free Cores
No
Needed # of instances = Initial # of instances
Calculated chunk size = ceil (Total # of frames / Needed # of instances)
Calculated chunk size < Initial chunk size
Yes
Actual chunk size = Initial chunk size
No
Actual chunk size = Calculated chunk size

Figure 4.5: The RCU working steps to calculate actual chunk size.

If (*Needed # of instances < New # of instances*) $\longrightarrow$

$$\therefore \textit{Final \# of instances} = \textit{Needed \# of instances} \tag{4.10}$$

else,

$$\textit{Final \# of instances} = \textit{New \# of instances} \tag{4.11}$$

8. In some cases, number of frames per individual video is not divisible by the chunk size we calculated. So, the last video chunk size could be less than the actual size. This makes the calculation of needed number of instances based on total number of frames of all video files in videos list is not completely accurate. As a double check, we recalculate the needed number of instances per video file and sum the total number of instances, then compare it with the value calculated in step 7 and select the greater one.

   Start with Total # of instances = 0,
   For(video in videos list):{

$$\textit{\# of instances per video} = \left\lceil \frac{\textit{Total \# of frames per video}}{\textit{Actual chunk size}} \right\rceil \tag{4.12}$$

$$\textit{Total \# of instances} += \textit{\# of instances per video} \tag{4.13}$$

   }

   If (*Final # of instances < Total # of instances*) $\longrightarrow$

$$\therefore \textit{Actual \# of instances} = \textit{Total \# of instances} \tag{4.14}$$

   else,

$$\textit{Actual \# of instances} = \textit{Final \# of instances} \tag{4.15}$$

9. Final values are: *Actual chunk size and Actual # of instances*

The entire algorithm designed to calculate the number of bolt instances is shown in Figure 4.6.

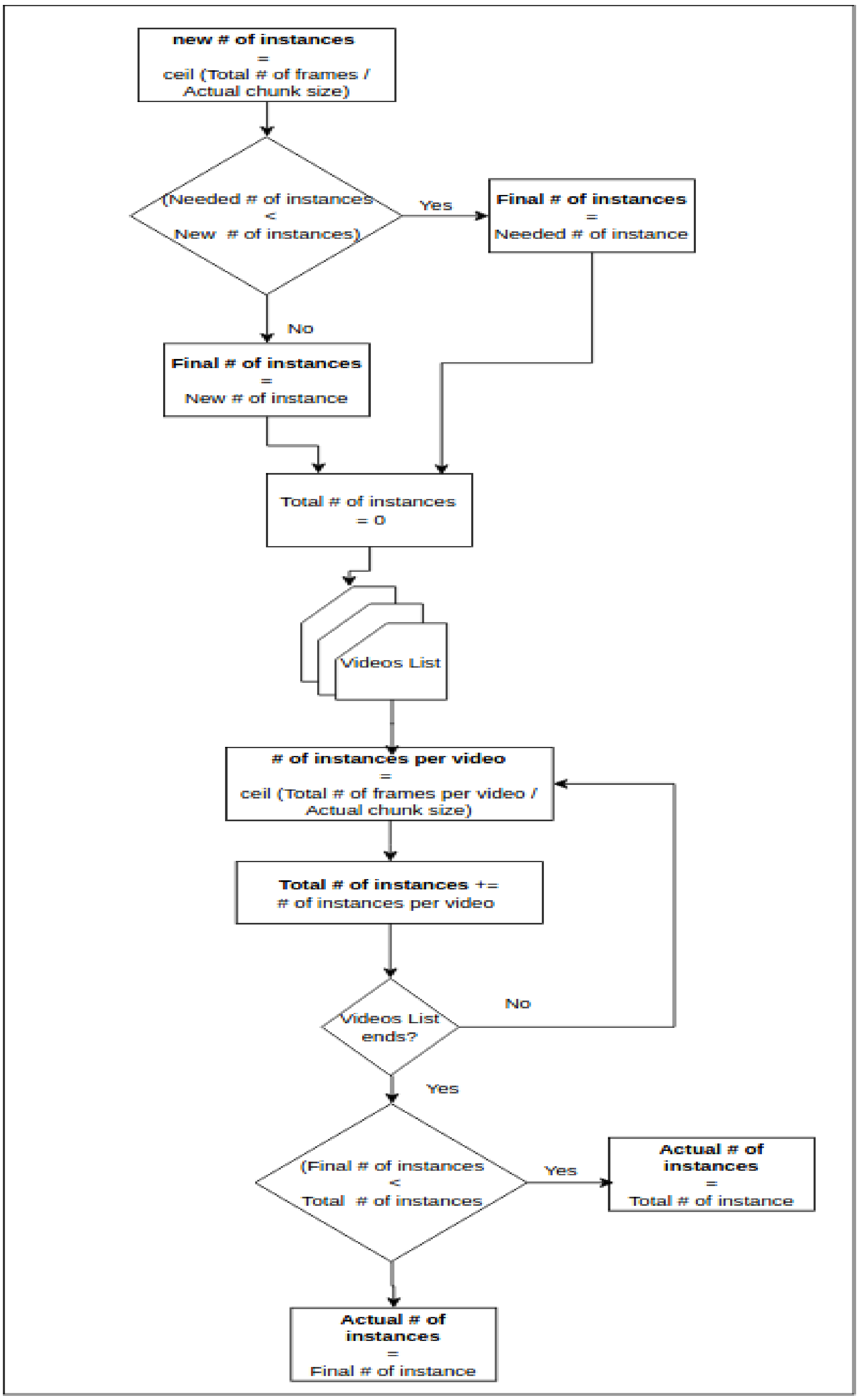


Figure 4.6: The RCU calculation of actual needed number of bolt instances.

4.3.2 Data Splitting and Feed unit (DSFU)

The Data Splitting and Feed Unit (DSFU) is responsible for splitting the input video files into equally sized small chunks where the chunk size is the actual chunk size calculated by the RCU as shown in Figure 4.7. Then these video chunks are fed frame by frame ordered to Storm spouts for processing. We give a unique ID to each video chunk which is the original video file name plus a numerical suffix that numbers each chunk. The chunks numbering starts from 0 in order of their presence in the original file itself. This unique chunks ID will be used later in grouping data probably within Storm and collecting the results of each chunk individually. Numbering of frames within each chunk is kept the same as their numbering in the original video file. For example; if the chunk size equals 1000 frames, then the first chunk will have frames numbered from 0 to 999 and the second chunk frames are from 1000 to 1999 and so on.

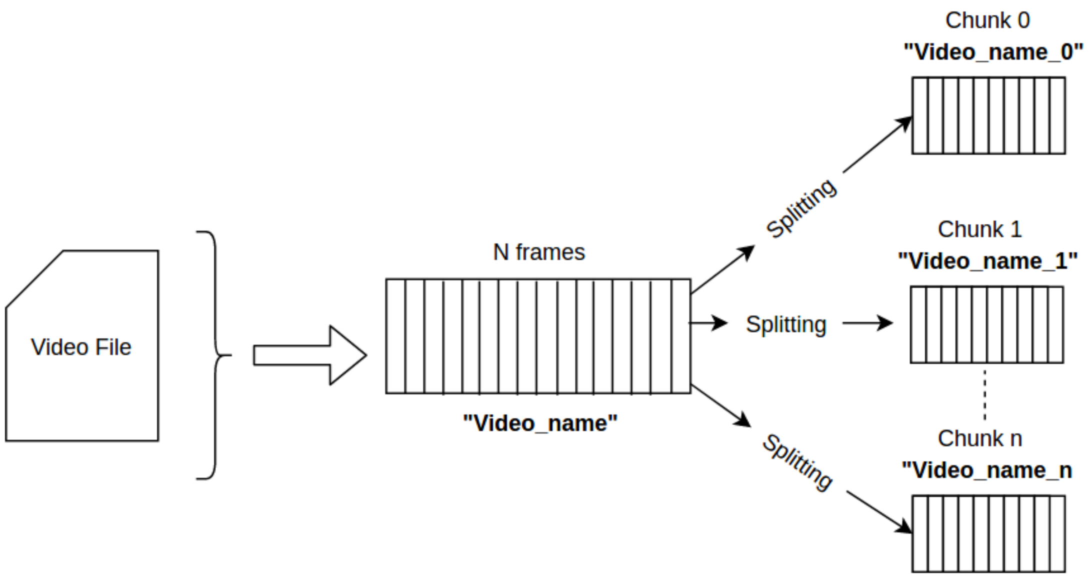


Figure 4.7: Splitting video files in DSFU.

Video files compression techniques depend on the fact that all the adjacent sequencing video frames have a relation and any frame can be predicted by knowing the previous frames preceding it. The first frame to appear is called an Iframe (Intra-coded picture) which is a self contained frame that will be used to predict the upcoming frames. The I-frame has all the needed information to know its content and reconstruct the next frames. The upcoming frames use the I-frame as a reference by observing the differences that occur in the image compared to I-frame. These frames are called P-frames (Predicted frames). Some frames can use the previous and next frames to it in order to predict its content, they called B-frames (Bidirectional predicted frame) [70, 71]. Each video file has many GOPs (Group of pictures) where each GOP consists of one I-frame and multiple of P and B frames [72].

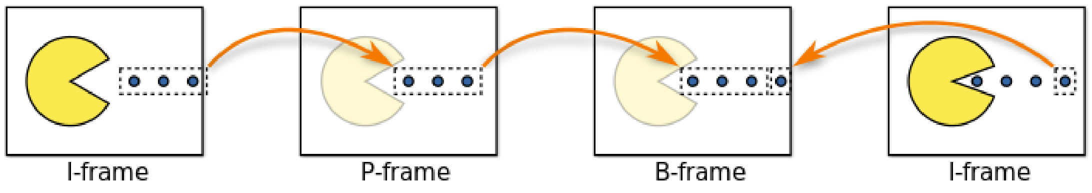


Figure 4.8: Illustration of video frames sequence consisting of I, P and B frames [70].

In our data splitting and feed unit, we take the actual chunk size calculated as a reference when splitting the video although we try to make sure that the first frame in each chunk is an I-frame. This is important as each chunk will be decoded and processed independently from other chunks and the info of an I-frame is needed to decode the other frames in the chunk.

The video chunks are sent to Storm spouts to be processed in the processing bolt. We send all the chunks frames simultaneously to the spout queue such that the first frames of each chunk are added to the spout receiving queue in almost the same time, then the second frames and the remaining ones all in order till the end of all chunks. We called this splitting and receiving technique Parallel chunk-based splitting, see Figure 4.9. It enables Storm to process all the videos chunks in parallel along the entire available resources.

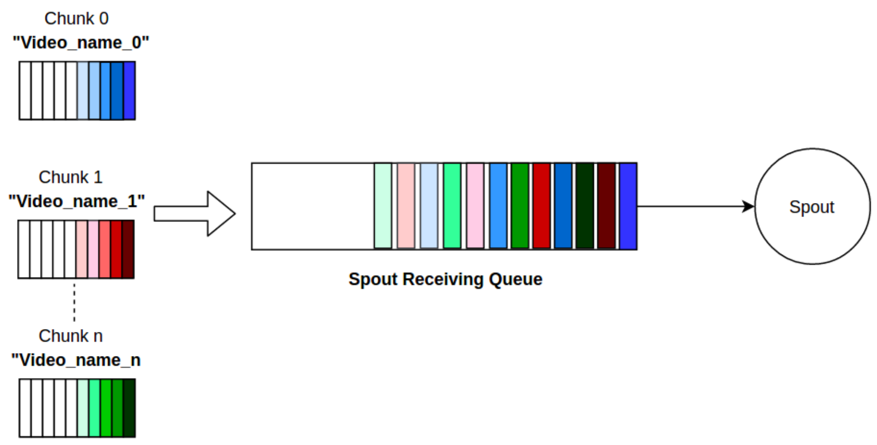


Figure 4.9: Sending chunks frames to spouts queue using parallel chunk-based splitting.

4.3.3 Decision Making and Resources Mapping Unit (DMRMU)

As mentioned before, Storm working topologies are constructed from group of spouts and bolts where the spouts are the data sources and bolts are the processing units. Bolts tasks are the actual processing units that are distributed among the working cluster machines. In case of sequential VP algorithms, it is required for all the frames that belong to one media source to be processed in the same bolt task while preserving their sequential order. So, after splitting video files into chunks in splitting unit, we need to group the entire frames of each chunk and emit them to one specific bolt task. Also we need to isolate the frames of each chunk and make sure that each bolt task receives all the frames from one chunk exclusively to maintain data consistency as shown in Figure 4.10.

To achieve these needs we implemented a custom Storm grouping using Storm built-in *Direct grouping* in addition to a developed decision mechanism that governs tasks allocation, data grouping and tuples emitting process. The Decision Making and Resources Mapping unit *DMRMU* is responsible for assigning bolt tasks to independent video chunks, grouping chunks frames, applying custom grouping rules and maintaining the assignment map between bolt tasks and chunks frames data during topology processing operations.

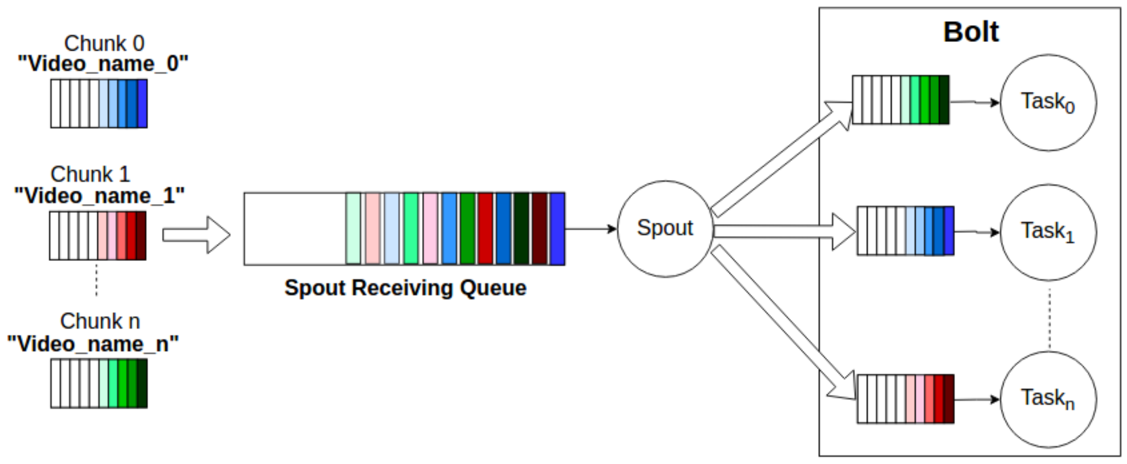


Figure 4.10: Data splitting and grouping in DCD-Framework.

We used Redis [73], an open-source in-memory database and messaging broker to create and store the assignment map and share these information between all working machines. Basiclly, Redis stores data as key-value pairs and supports various types of data structures such as strings, lists, hash tables, sets, sorted sets, geospatial data . . . etc. It supports many programming languages such as Java, Python, C++, Ruby, PHP and much more [74, 75]. In our development we used lettuce [76] which is a Redis Java client that is thread safe for all synchronous and asynchronous connections with Redis under Apache license. This suits our developed Storm applications that are written in Java. Also to execute complex queries and computations in Redis, we used Lua scripts [77] a lightweight and powerful scripting language that supports functional, object-oriented and procedural programming. We were able to write lua scripts in Java using Lettuce.

When the topology is submitted, it creates number of bolts tasks equals to the actual number of needed instances calculated in resources calculation unit. The tasks are distributed evenly between the cluster machines using Storm default fair scheduler. Based on the calculated actual chunk size, we limit the number of chunks to be submitted to one topology at a time to equal number of needed bolt tasks instances. This gives a one-to-one mapping between the submitted chunks and the running tasks. To distribute the video chunks over these tasks the DMRMU creates and maintains in Redis the following:

- **Tasks IDs queue:** A Redis queue contains all bolt tasks IDs in any order.
- **Tasks-to-Chunks assignment map:** A Redis hash map with <key, value> pairs represent $< \text{Task}_{\text{ID}}, \text{Chunk}_{\text{ID}} >$.

Figure 4.11 shows a sample view of Redis contents with some values as example.

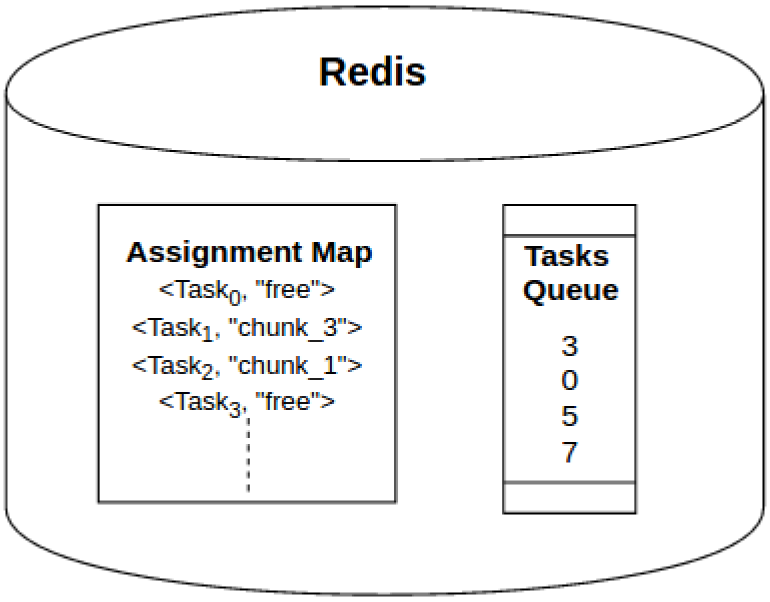


Figure 4.11: Redis sample view of Assignment Map and Tasks Queue.

The steps to create the *Tasks queue* and *Assignment map* during Storm topology creation are:

1. Each bolt task gets its unique ID in initialization step and pushes it to the top of tasks IDs queue in Redis.
2. A new record is created in the assignment hashmap in Redis with $< key, value >=<$ $\text{Task}_{\text{ID}}, "free" >$. *"free"* in record value means that this task isn't assigned yet to any chunk.

When the spout starts receiving chunks frames, it is required to assign one bolt task to each individual chunk and send the entire chunk frames exclusively to this bolt task in sequential order while preserving the frames in the same order the spout receives them from its queue. We proposed a methodology to utilize the created *Tasks queue* and *Assignment map* in order to perform the assignment operation successfully and route each frame to its destined task correctly. Also, we took into consideration the precautions to avoid resources racing between chunks.

Each Spout in the topology performs the following when it receives one video frame:

1. Extract the frame $\text{Source}_{\text{ID}}$ from the respective metadata sent with the frame. This frame belongs to the chunk with $\text{Chunk}_{\text{ID}} = \text{Source}_{\text{ID}}$
2. Check if there is an assigned bolt task for this chunk with Chunk_{ID};
   The spout sends a request to the developed Redis client in DMRMU to get any record in the Tasks-to-Chunks assignment map where the records *value* equals the current Chunk_{ID}.
   If yes, then there is an assigned bolt task for all the frames belonging to this chunk.

   - The DMRMU retrieves the associated records *key* which is the assigned Task_{ID} for this video chunk.
   - Change the spout emitting stream ID of this frame to equal Task_{ID}.

   If no, then this is the first frame of this chunk and we need to pick a free task and allocate this task to the current video chunk.

   - The DMRMU pops the first free Task_{ID} from the *Tasks queue* in Redis and returns it as the candidate free slot.

- Using Redis client, the *Assignment map* is updated by changing the *value* of the record $< key, value >=<$ Task$_{\text{ID}}$ = "$free$" $>$ to $< key, value >=<$ Task$_{\text{ID}}$ = Chunk$_{\text{ID}}$ $>$ where the Chunk$_{\text{ID}}$ equals the Source$_{\text{ID}}$ of the current frame.
- Change the spout emitting stream ID of this frame to equal Task$_{\text{ID}}$.
- Any new frames belonging to this chunk will be sent to the same bolt task with the Task$_{\text{ID}}$ selected in this stage

We used a Redis queue to store and retrieve the task ID in order to limit the access of this queue to one requesting chunk at a time. Redis queue limits the access to pop values from it to one request at a time. This eliminates the chance of selecting the same task by many chunks and avoid any case where the chunks race over the available resources. Processing resources allocation and decision process for each video frame is depicted in Figure 4.12.

Storm spout declares its output streams as *Direct streams* to have the ability of deciding the receiving bolt task for each chunk. Using the Task$_{\text{ID}}$ assigned to each Chunk$_{\text{ID}}$, a spout alters the receiving Task$_{\text{ID}}$ using *emitDirect* method which is a built-in method in Storm. This is a custom implementation of Storm streams *Direct Grouping*. Hence, incoming data streams tuples are no longer distributed randomly to the consuming bolts but the distribution process complies with the *Assignment map* and DMRMU policies.

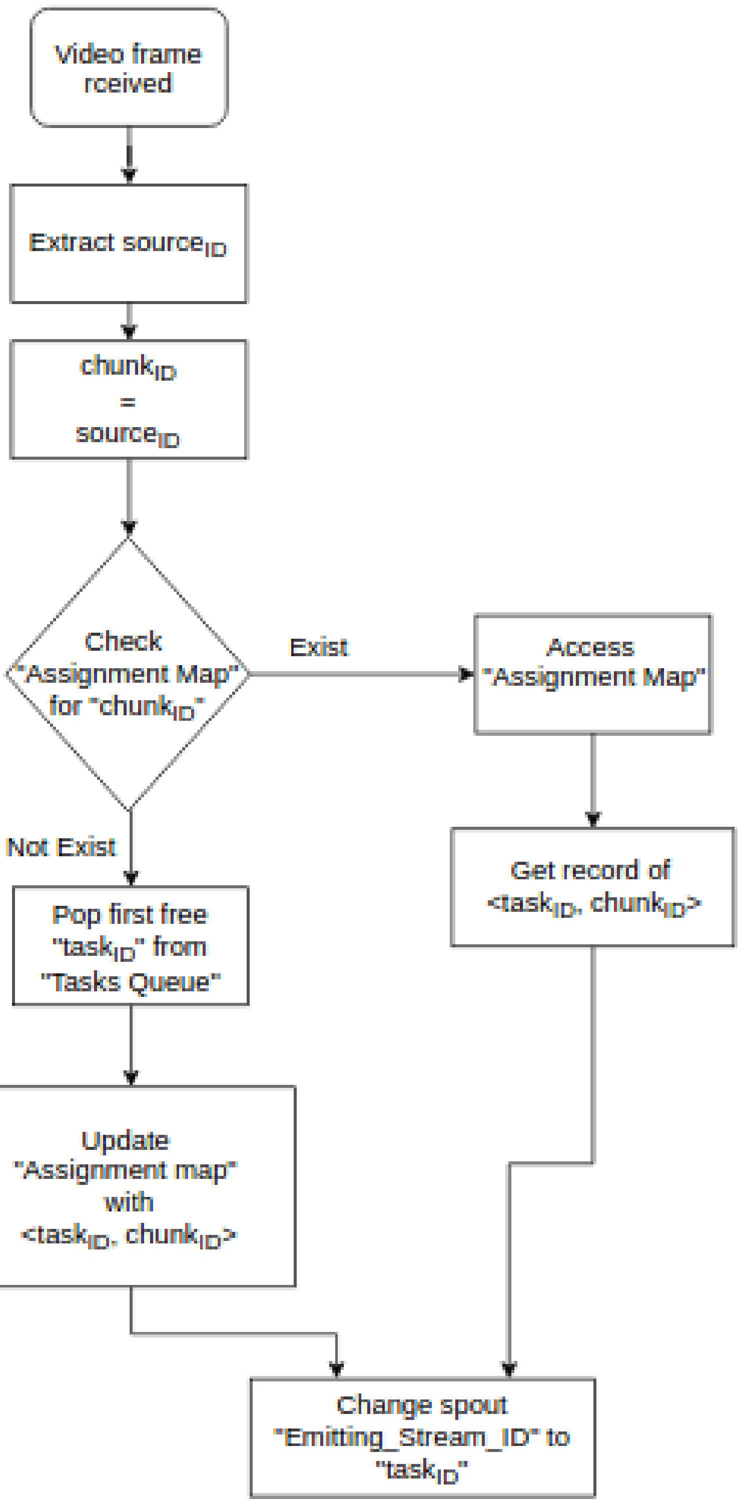

Video frame rceived
Extract source$_{ID}$
chunk$_{ID}$ = source$_{ID}$
Check "Assignment Map" for "chunk$_{ID}$"
Exist
Access "Assignment Map"
Get record of <task$_{ID}$, chunk$_{ID}$>
Not Exist
Pop first free "task$_{ID}$" from "Tasks Queue"
Update "Assignment map" with <task$_{ID}$, chunk$_{ID}$>
Change spout "Emitting_Stream_ID" to "task$_{ID}$"

Figure 4.12: Selecting Task$_{ID}$ for each video frame.

4.4 Proposed Framework Key Competencies

The DCB-Framework proposed in this thesis has generic architecture and working logic. Most of sequential CV and VP algorithms can be ported in Apache Storm for processing using DCB-Framework. Commodity servers and hardware with reasonable memory and number of CPU cores can work fine with DCB-Framework. That is why there is no need for specific hardware requirements to utilize this processing framework and gain processing speedup. Despite that the main target of this framework is to enhance and parallelize the processing of sequential VP algorithm, the DCB-Framework is generic and can adopt parallel algorithms as well.

Another important advantage of DCB-Framework is the ability to process multiple video files simultaneously while preserving data consistency and isolation within Storm processing bolts. This leverages the horizontal scalability of Storm big-data framework efficiently. Data distribution within our proposed framework makes it possible to perform sequential VP in the scope of one video chunk through big-data processing frameworks which is an added point to this thesis concept.

4.5 Conclusion

In this chapter we proposed a new processing framework that enables us to port sequential CV and VP algorithms within big-data processing frameworks. The developed algorithm basic idea is to split each video file into sub chunks and distribute the processing of these chunks over the available processing CPU cores under controlled data grouping and distribution logic. Apache Storm was the selected big-data tool to implement the DCB-Framework. Storm was the best fit to this thesis work due to its ability to provide real-time processing while achieving low latency and fault tolerance. The core points and new system architecture of the proposed framwork were discussed in details. In the next chapter we review the experimental results done to evaluate the DCB-Framework and conclude to the enhancements and performance gain achieved.

Chapter 5: Experimental Results

5.1 Introduction

In this chapter several experimental tests were performed to evaluate the performance of the proposed DCB-Framework against different sequential CV algorithms. The selected use cases are *Video Summarization (VS)*, *License Plate Recognition (LPR)*, *Face Detection* and *Heatmaps*. Then, we illustrate the hardware setup and Storm topology structure used to conduct these tests. Various evaluation metrics are taken into consideration to prove the processing gain that our proposed framework can reach. These tests are done over 1 video file for each CV algorithm then we selected *Face Detection* for testing against multiple input video files.

5.2 Evaluation Process Setup

5.2.1 Hardware Setup

A Microsoft Azure online virtual machine was used to conduct the evaluation testing trials. The virtual machine has 32 Intel CPU cores, 256 Gigabyte memory and 1 Terabyte hard disk. Ubuntu version in the testing machine is 16.04. Also it has Apache Storm version 1.0.1, Redis version 3.2.8 and Apache Thrift version 0.9.3 installed.

5.2.2 Storm Topology Structure

Storm topology for VP can run on two setups: standalone or distributed cluster. In standalone setup we have one physical machine that has powerful processing capabilities and hold the responsibilities of both master and slave machines. In distributed setup we have one physical master machine with minimal hardware requirements in addition to one or more powerful slave machines for processing. In our evaluation process we used the standalone setup due to the limitation of the available powerful machines for testing. However the algorithm is general and can be applied for distributed cluster setup with no change in the logic of topology architecture.

Generally, in any setup we need only one Storm nimbus daemon that is used to submit and monitor cluster running topologies as described in chapter 2. For each slave machine we need one Storm supervisor for Storm tasks scheduling. We limit one spout with one running task per physical salve machine that is responsible for reading input video frames sent over thrift and distribute them for processing in all existing bolt tasks across the entire cluster. Only one bolt will be submitted in the processing topology with multiple running tasks. Number of bolt tasks in the submitted Storm topology is limited by the number of video chunks that needed to be processed simultaneously.

Let's say we have 3 input videos each one is divided into 3 chunks, which means that the processing bolt needs to have 9 individual bolt tasks that contain the processing engine loaded and ready to process different data. Each spout or bolt tasks runs in a separate Storm executor. Thus number of topology executors equals to:

$$\#\ of\ Topology\ Executors = \#\ of\ Spouts + \#\ of\ Bolt\ Tasks \tag{5.1}$$

We run a separate Storm worker for handling supervisor duties in each physical slave machine while each spout or bolt task has its own worker. So number of Storm workers in VP topology equals:

$$\#\ of\ Topology\ Workers = \#\ of\ Slave\ Machines + \#\ of\ Spouts + \#\ of\ Bolt\ Tasks \tag{5.2}$$

Storm component parallelism hint equals to initial number of executors assigned to this component in topology submission. Moreover the topology total parallelism factor equals to sum of all components executors in the topology. One slave machine in VP topology components is shown in Figure 5.1.

Storm configurations to tune spouts automatic back pressure [78, 79] are listed in Table 5.1. Automatic back pressure is a practical feature was added to Storm starting from version 1.0.0 [80]. It aims to throttle the spout flow of sending new tuples for processing in case of bolt congestion. This feature depends on the state of the bolt receiving queues whether they are empty or full. We have two important watermarks that tune the speed of spout data tuples sending; high and low watermarks. These watermarks define the percentage of bolt tasks occupation size. When high watermark is reached, the spout sending rate is slowed down until the bolt consumes group of pending data tuples and the queue is emptied till reaching low watermark. At low watermark, the spout is allowed to resume sending data tuples to the topology tasks again. The parameters used to tune Storm back pressure are presented in Table 5.1. Back pressure feature allows us to decrease number of failed/non-acked data tuples and enhance data complete latency and throughput.

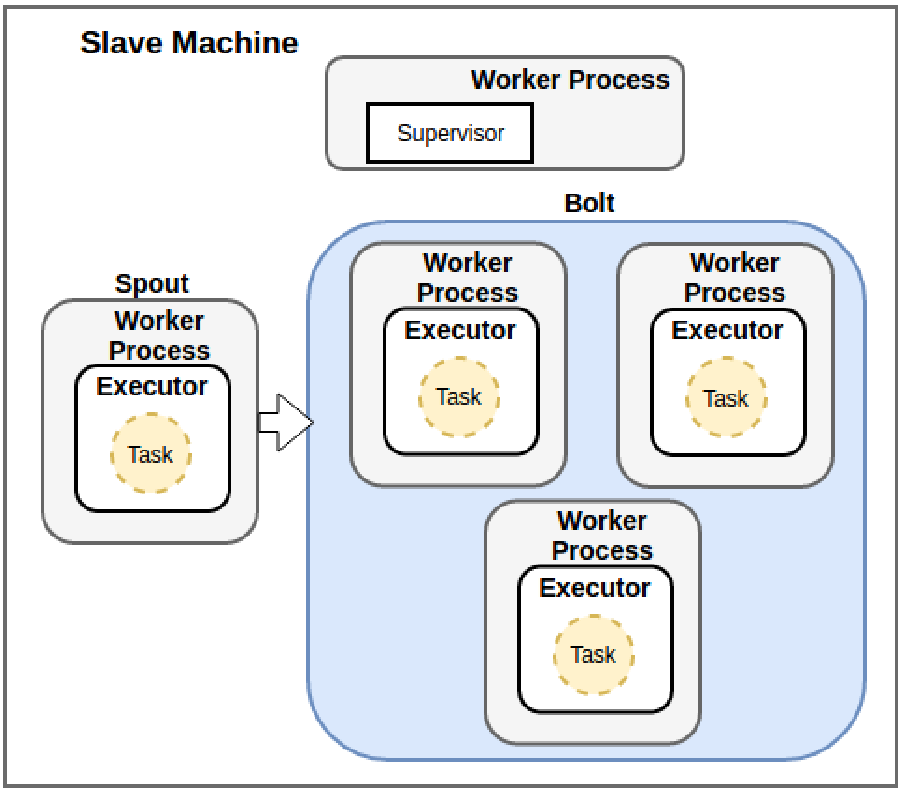


Figure 5.1: Storm slave machine components for VP.

Table 5.1: Storm configuration for automatic back pressure.

Configuration	Value	Function
topology.max.spout.pending	1000	Defines max number of pending messages and spout sending queue
backpressure.disruptor.high.water mark	0.9	Send the « full » signal, and throttle the spout when the receive buffer of the bolt is 90% full
backpressure.disruptor.low.waterm ark	0.6	Send the « not full » signal, and start the spout again when the receive buffer of the bolt drops below 40% of capacity
topology.executor.receive.buffer.siz e	1024	Defines how many messages max we can have pending for a bolt or spout
topology.executor.send.buffer.size	512	Defines how many messages max can be buffered for emitting

5.2.3 Evaluation Metrics

In order to evaluate the presented algorithm in this thesis, we collected different performance metrics during the testing runs. First, we collected the total processing time of each testing video file in all use cases. The total processing time is the difference between the timestamp of first video frame received by the spout and the timestamp of last processed frame in bolt tasks. Then, we calculated how many times speed up we achieved compared to the default sequential processing. Let's say T_s is the processing time of the video file in sequential mode and T_c is the processing time in testing chunk-based case. The speed up gain is:

$$\textit{Speed Up} = \frac{T_s}{T_c} \tag{5.3}$$

. The speed up of the sequential processing is considered 1.

Using Storm web UI, we captured an important metric called *Complete Latency* which is the total duration between spout emitting the tuple and the processing tuple tree being completed and acked successfully. That includes the waiting time the frame's data tuple spends in the spout sending queue waiting its turn to be emitted for processing. The complete latency reflects the slowest path of the topology tree. It is also a reflection of how efficient the back pressure tuning is and if the available resources are enough to process frames in parallel with minimal waiting time. We implemented a custom Storm metrics that measures the average of how many frames per second (fps) the spout is receiving and sending frames to the topology and the average bolt task processing fps.

5.3 Evaluation Test Cases

In this section we assess the performance of DCB-Framework for different sequential CV popular algorithms. The algorithms under consideration are *Video Summarization (VS)*, *License Plate Recognition (LPR)*, *Face Detection* and *Heatmaps*. For testing we used 1 hour video file of moderate activity for each processing algorithm. The specifications of these video files are listed in Table 5.2. Noting that we scale the input video files frame rate to equals 15 fps. That removes the redundancy and eliminates the processing of unnecessary frames. For example if we have a 30 fps input video file, we will drop almost 15 frame from each second to decrease number of processed frames and unify the frames rate for all testing videos. When number of chunks = 1, the algorithm runs sequentially over the video frame by frame. Chunk mode is considered for number of chunks ≥ 2.

Table 5.2: Testing video specifications for each VP algorithm.

Algorithm	Video Duration	Original number of frames	Video fps	Final Number of Frames	Resolution	Extension
Video Summarization	1:00:00.12	57878	16.08	54002	640x540	avi
License Plate Recognition	1:00:00	108000	30	54000	768x432	avi
Face Detection	1:00:00	90000	25	54000	768x432	avi
Heat Map	0:55:25.6	83140	25	49884	640x480	mp4

The original total number of frames per video $\mathrm{N_{Original}}$ is calculated as follows:

$$\mathrm{N_{Original}} = \mathrm{D_{seconds}} \times \mathrm{Videos_{fps}} \tag{5.4}$$

Where $\mathrm{D_{seconds}}$ is the video duration in seconds and $\mathrm{Videos_{fps}}$ is the original video fps. The final number of frames per video $\mathrm{N_{Final}}$ after scaling the fps to 15 is:

$$\mathrm{N_{Final}} = \mathrm{N_{Original}} \times \frac{15}{\mathrm{Videos_{fps}}} \tag{5.5}$$

5.3.1 Video Summarization

Video Summarization algorithm is used to take hours of video files as input, pass by it frame by frame to extract all the important events and produce an output summarized video that is a comprehensive version of the original one. The duration of the output video is in minutes which is small compared to the original input video. The output summarized events extend to anything that is moving in the scene (e.g. cars and people . . . etc.). Sample input and output screenshot for video summarization algorithm is show in Figure 5.2.

The video frames are fed in succession to the algorithm where objects that are presented in those frames are extracted and tracked throughout the rest of frames. A post-processing step is applied to eliminate invalid or short-time appearing objects. In order to generate a meaningful summary, those objects are drawn in a particular order matching that of the input video. Yet multiple objects can be drawn in a single frame based on user input to squeeze the summary duration. All objects are then fused in one output video labeling each with its original timestamp. Video summarization plugin depends on these libraries; OpenCv 3.3 and Bgslibrary.

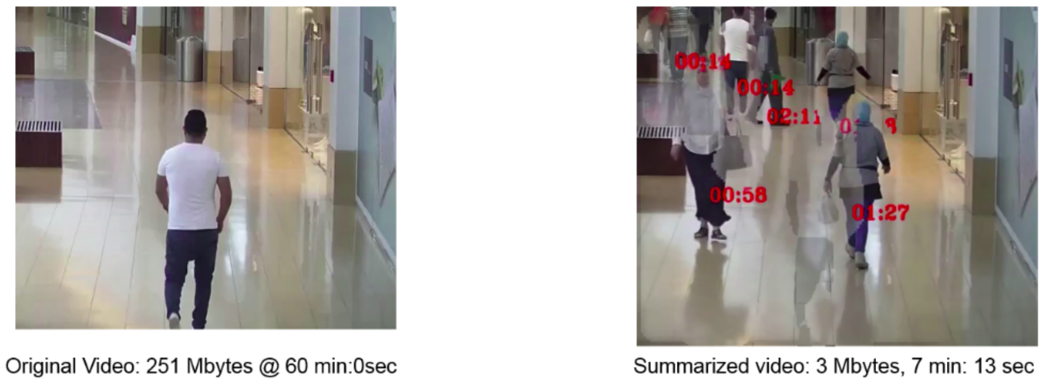

Original Video: 251 Mbytes @ 60 min:0sec
Summarized video: 3 Mbytes, 7 min: 13 sec
00:14
00:14
02:11
00:58
01:27

Figure 5.2: Video Summarization algorithm output sample.

In order to test the enhancement of DCB-Framework processing over video summarization algorithm, we used the 1 hour duration testing video with specifications defined in Table 5.2. We wanted to examine the effect of splitting this video file over different number of video chunks that are processed in individual bolt tasks. The collected performance evaluation metrics are listed in Table 5.3.

In Figure 5.3, Figure 5.4 and Figure 5.5 we plotted the total processing time, the processing speed up and complete latency of video summarization against number of chunks. It is shown that by increasing number of chunks, the total processing time of testing video decreases and we got a maximum speed up of around 5x starting from using 13 chunks per video compared to processing it sequentially. The spout complete latency decreases for video summarization as we reach the saturation point at almost 13 chunk per video file.

Table 5.3: Video summarization algorithm evaluation metrics.

# Chunks/ # Bolt tasks	Processing Time in Minutes	Processing Speed up	Spout avg. fps	Bolt task avg. fps	Complete Latency (s)
1	44.60	1.0	22.51	22.12	31.40
2	26.02	1.7	37.58	19.30	28.40
3	18.18	2.5	51.00	17.10	27.02
4	15.85	2.8	62.30	15.90	26.24
5	14.17	3.1	71.68	14.60	25.23
6	12.72	3.5	79.80	13.50	24.20
7	11.02	4.0	87.09	12.61	21.94
8	10.30	4.3	92.85	11.40	20.08
9	9.87	4.5	99.60	11.00	18.60
10	9.45	4.7	104.36	10.60	16.40
11	9.12	4.9	108.65	9.70	12.20
12	8.81	5.0	108.45	9.20	5.60
13	8.72	5.1	108.60	8.70	2.40
14	8.72	5.1	108.00	7.50	1.24
15	8.72	5.1	108.00	7.00	0.85

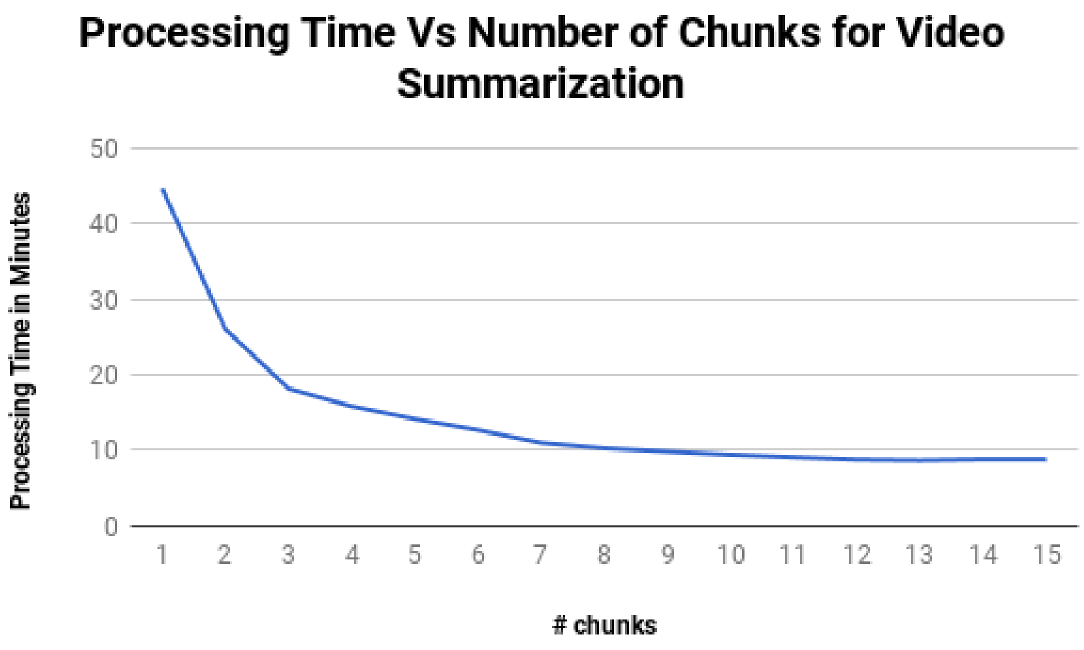


Figure 5.3: Total processing time of testing video for video summarization.

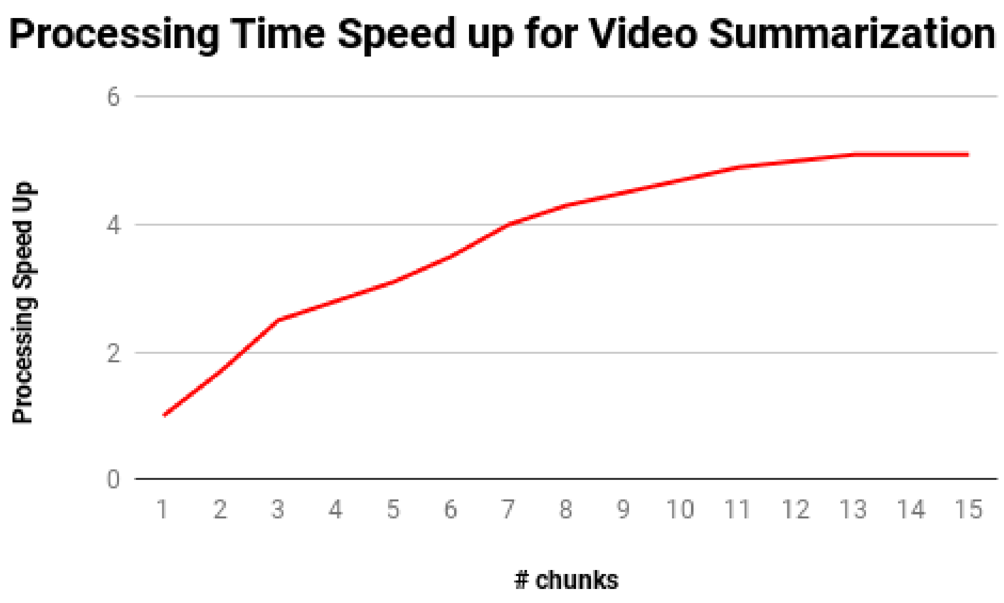


Figure 5.4: Processing time speed up of testing video for video summarization.

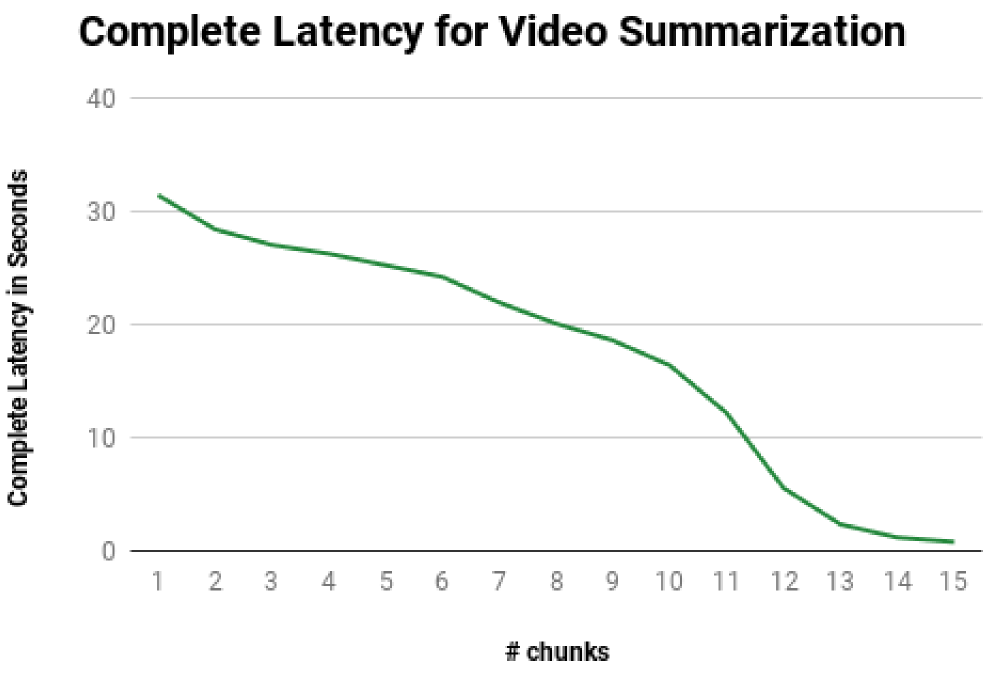


Figure 5.5: Complete latency of testing video for video summarization.

5.3.2 Face Detection

The Face Detection VP algorithm aims to find all possible faces in an image or a video frame. It takes a video frame as an input. If the input frame size is larger than 150 x 150 pixels the width and height of the frame are resized by 1/5. Then the original (or resized) frame is forwarded to a neural network that detects the faces in the frame. The network is a C++ implementation of the multitask Convolutional Neural Network (CNN) using OpenCv library for image processing and Caffe framework for neural network processing. Example of face detection result for one video frame is shown in Figure 5.6.

The performance evaluation of face detection processing against different number of parallel video chunks is listed in Table 5.4. These values are for the 1 hour testing video described in Table 5.2. The evaluation metrics parameters graphs are shown in Figure 5.7, Figure 5.8 and Figure 5.9. It is noted that the total VP time decreases by increasing number of video chunks till we reach a max speed up of 7.8x starting from 26 chunks per video file. This reflects the complete latency decreasing to lower values when the processing speed up approaches the saturation point.

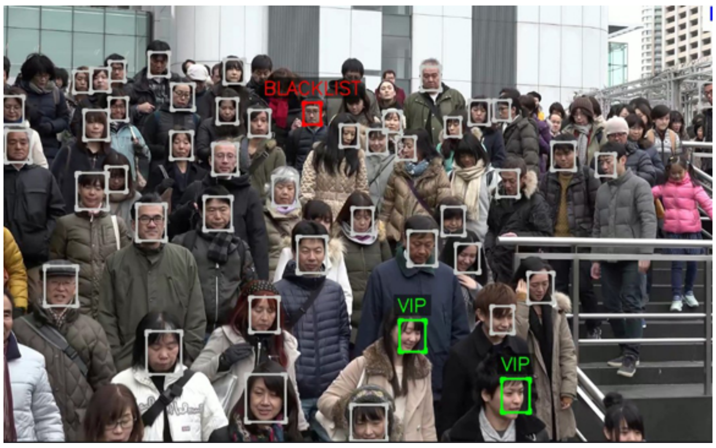


Figure 5.6: Face detection algorithm output sample [81].

Table 5.4: Face detection algorithm evaluation metrics.

# Chunks/ # Bolt tasks	Processing Time in Minutes	Processing Speed up	Spout avg. fps	Bolt task avg. fps	Complete Latency (s)
1	38.75	1.0	23.4	23.0	31.2
2	23.08	1.7	39.4	19.3	24.6
3	17.33	2.2	52.9	17.2	18.9
4	15.30	2.5	60.0	14.4	17.0
5	13.28	2.9	69.9	13.7	14.9
6	11.46	3.4	80.1	12.8	12.5
7	11.13	3.5	84.1	11.9	12.1
8	10.37	3.7	89.5	11.1	11.5
9	9.65	4.0	96.2	10.4	10.1
10	9.08	4.3	102.8	9.8	9.7
11	8.45	4.6	109.8	9.4	9.2
12	7.98	4.9	115.4	9.3	8.7
13	7.53	5.1	123.7	9.3	8.4
14	7.20	5.4	130.0	9.0	7.6
15	6.87	5.6	136.9	9.0	7.3
16	6.57	5.9	143.3	8.8	6.8
17	6.32	6.1	148.9	8.2	6.7
18	6.12	6.3	153.2	8.1	6.2
19	5.85	6.6	161.0	8.3	5.9
20	5.58	6.9	168.4	8.0	5.4
21	5.43	7.1	173.0	7.8	5.0
22	5.23	7.4	180.2	7.8	3.6
23	5.13	7.6	181.9	7.8	2.0
24	5.13	7.6	181.9	7.1	0.9
25	5.00	7.7	181.8	6.9	0.4
26	4.98	7.8	181.8	6.7	0.3
27	4.98	7.8	181.7	6.5	0.2
28	4.95	7.8	181.6	6.2	0.2

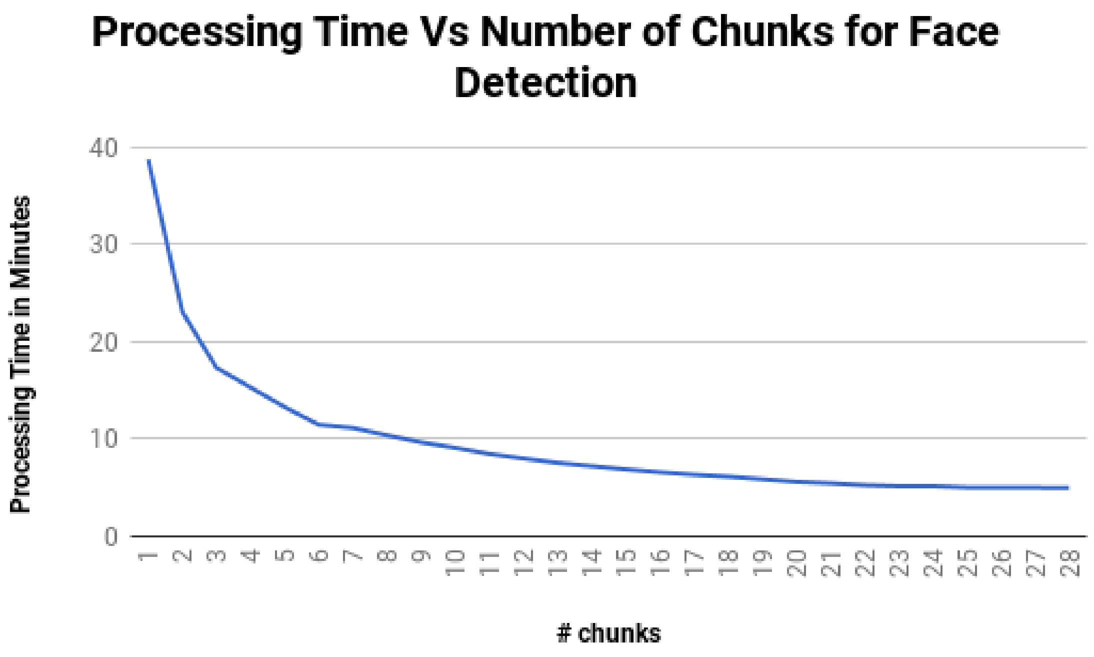


Figure 5.7: Total processing time of testing video for face detection.

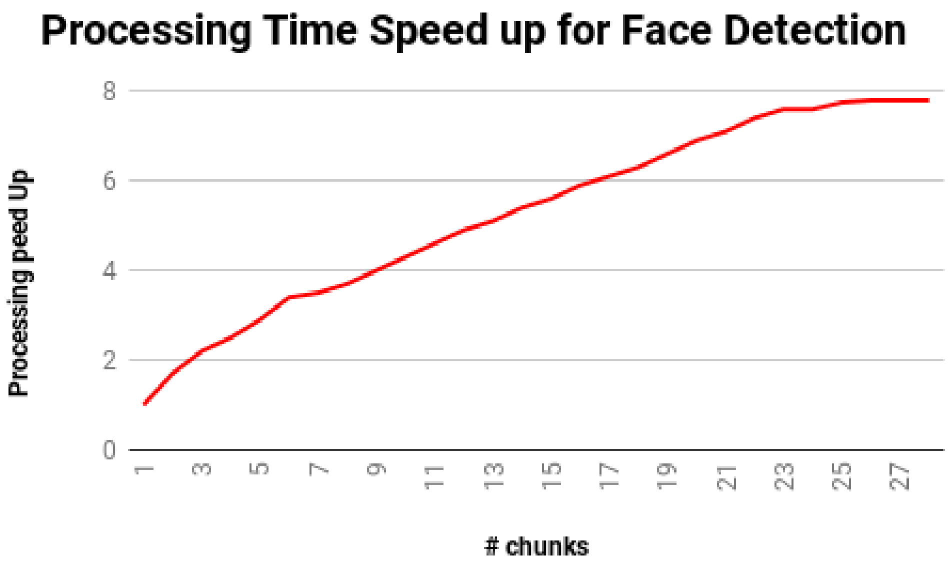


Figure 5.8: Processing time speed up of testing video for face detection.

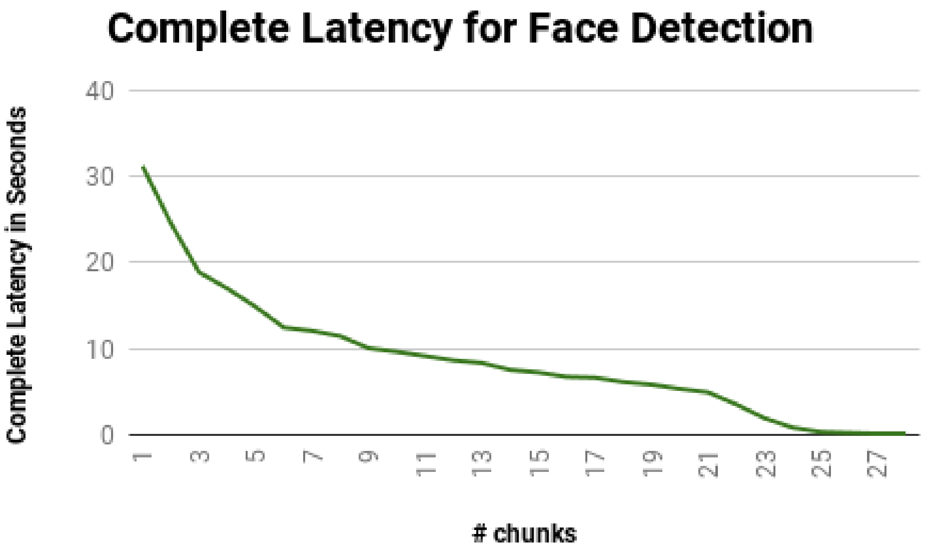


Figure 5.9: Complete latency of testing video for face detection.

5.3.3 License Plate Recognition

The License Plate Recognition (LPR) is used basically to extract the vehicles license plates information from recorded surveillance videos. The vehicles images are detected from video surveillance cameras that may be installed in streets, facilities gates or parking areas. The algorithm takes the video frames one by one and pass it through sequence of functions. First an OpenCV cascade classifier is used for detecting the license plate in the video frame. If a specific ROI is defined for this algorithm, only part of the frame will be processed which is the part that highly contain a license plate. After plate detection, the frames pass through a neural network to perform Optical Character Recognition (OCR). The OCR function is to extract each letter and number from the license plate to get the license information. LPR may have multiple classifiers for different countries plates standards. A detected car plate out of LPR algorithm is shown in Figure 5.10

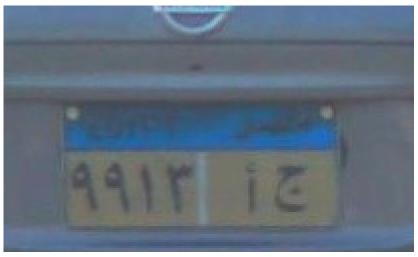


Figure 5.10: LPR algorithm output sample.

Table 5.5 shows the evaluation metrics captured from running LPR testing video specified in Table 5.2 using DCB-Framework processing for different number of chunks. From Figure 5.11 we found that LPR total processing time decreases remarkably by increasing number of splitted video chunks. We achieved a max processing speed up of about 4.7x as depicted in Figure 5.12. The spout frame complete latency decreases to a negligible value when we reach the speed up saturation point which is about 10 chunks for 1 hour testing video, see Figure 5.13.

Table 5.5: License plate recognition algorithm evaluation metrics.

# Chunks/ # Bolt tasks	Processing Time in Minutes	Processing Speed up	Spout avg. fps	Bolt task avg. fps	Complete Latency (s)
1	51.65	1.0	17.80	17.3	39.50
2	29.08	1.8	31.50	15.3	31.90
3	20.85	2.5	44.04	14.3	22.80
4	17.08	3.0	53.70	13.0	18.70
5	14.65	3.5	62.50	12.1	15.80
6	13.18	3.9	69.70	11.2	14.00
7	12.55	4.1	73.23	10.1	13.30
8	11.28	4.6	81.09	9.8	5.80
9	11.10	4.7	81.30	8.8	3.20
10	11.08	4.7	81.43	7.9	0.83
11	11.08	4.7	81.30	7.2	0.66
12	11.08	4.7	81.30	6.6	0.60

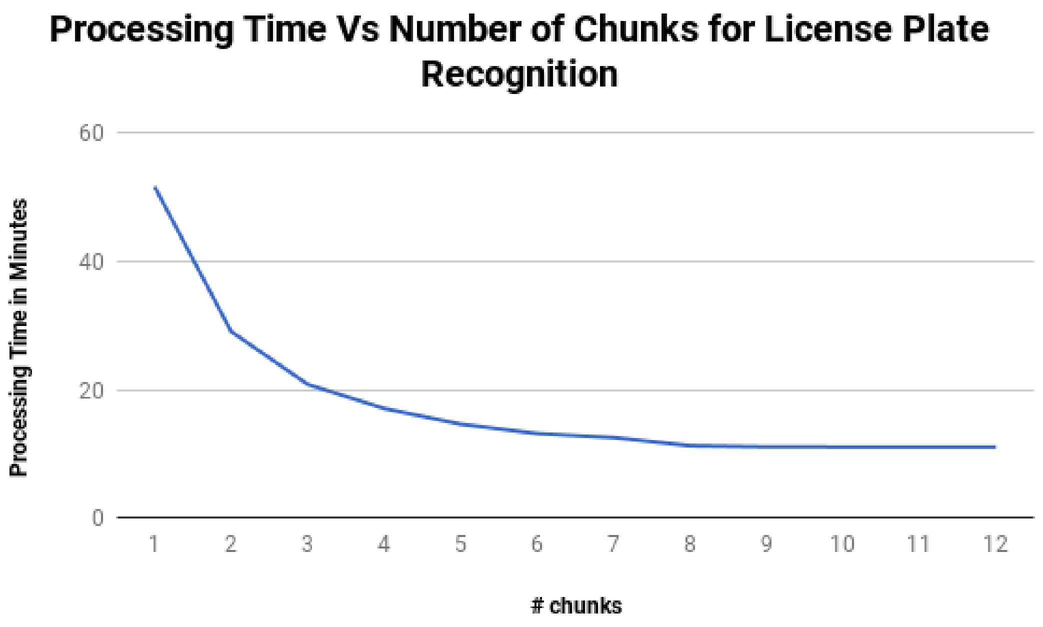


Figure 5.11: Total processing time of testing video for license plate recognition.

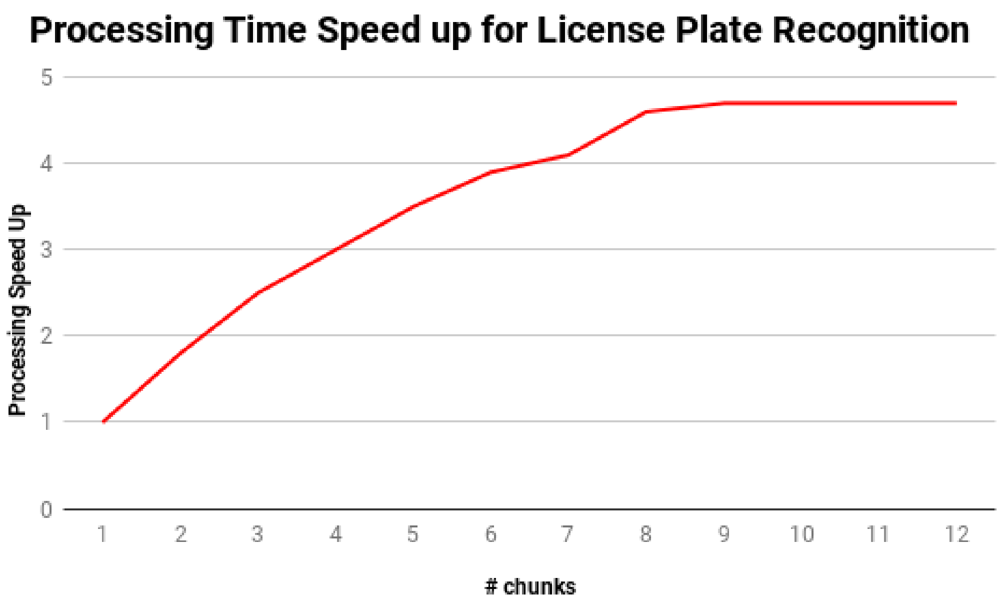


Figure 5.12: Processing time speed up of testing video for license plate recognition.

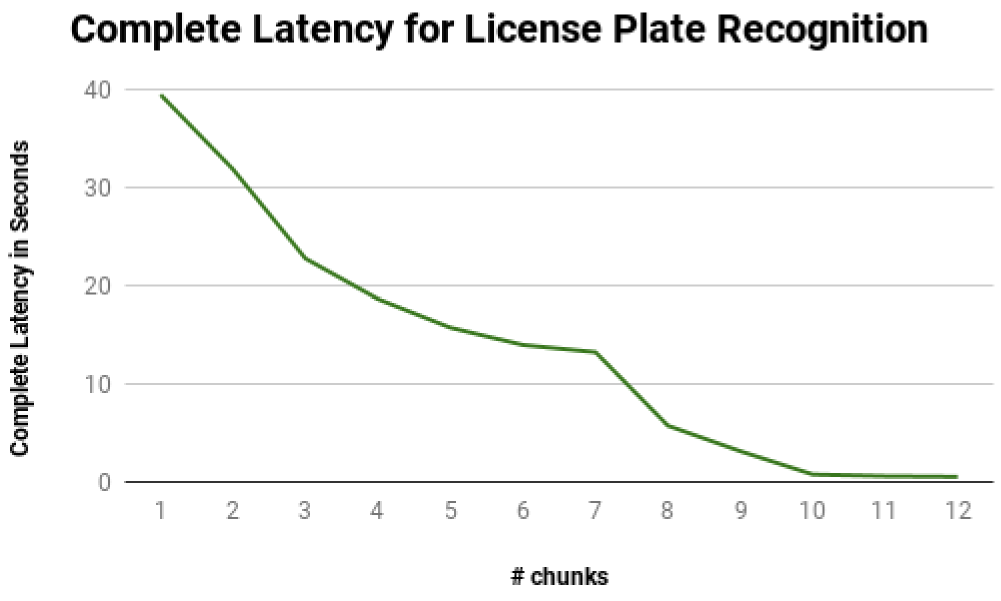


Figure 5.13: Complete latency of testing video for license plate recognition.

5.3.4 Heatmaps

Heatmaps algorithm is used for group activity detection in captured videos in order to show the crowd distributions at different designated areas. The heatmaps algorithm first models scene changes by subtracting background using sequence of video frames which enables detecting areas of movement. Then the algorithm applies filtration and morphological operations in order to remove noise from the results. Using result filtration over sequence of frames, algorithm generates 2-D histogram for movement areas. Finally, clustering is applied to classify the 2-D histogram into different density ranges with their ratios. Figure 5.14 shows sample of Heatmaps expected output. Heatmaps algorithm is implemented using OpenCV/C++.

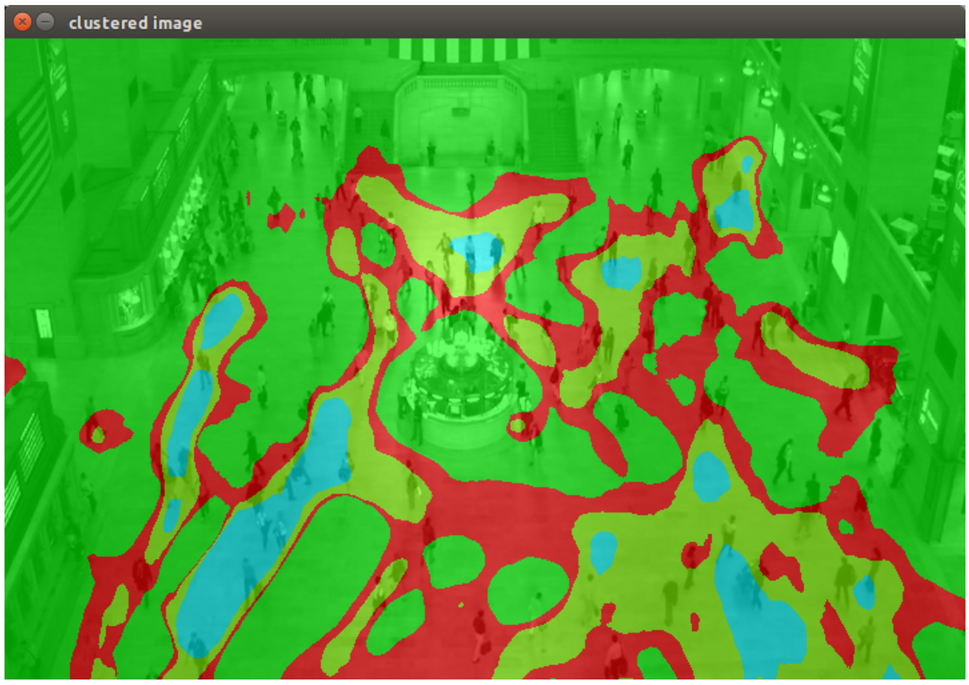


Figure 5.14: Sample output of heatmaps algorithm.

From heatmaps evaluation metrics captured in testing trials we found that the processing total time decreases by increasing number of chunks per testing videos. Referring to Table 5.6, the maximum processing speed up we can get is 2.6x times from running the heatmaps sequentially over 1 hour testing video. This speed up occurs at the algorithm saturation point which is almost in 5 chunks per video as show in Figure 5.15 and Figure 5.16. This processing speed up comes along with noticeable decrease in spout complete latency as depicted in Figure 5.17.

Table 5.6: Heatmaps algorithm evaluation metrics.

# Chunks/ # Bolt tasks	Processing Time in Minutes	Processing Speed up	Spout avg. fps	Bolt task avg. fps	Complete Latency (s)
1	26.250	1.0	31.95	31.24	21.60
2	16.620	1.6	51.11	24.80	19.50
3	12.580	2.0	67.57	21.80	14.50
4	10.900	2.4	77.90	18.60	11.70
5	10.083	2.6	82.40	16.10	0.340
6	10.083	2.6	82.00	13.30	0.082
7	10.083	2.6	82.00	11.40	0.074

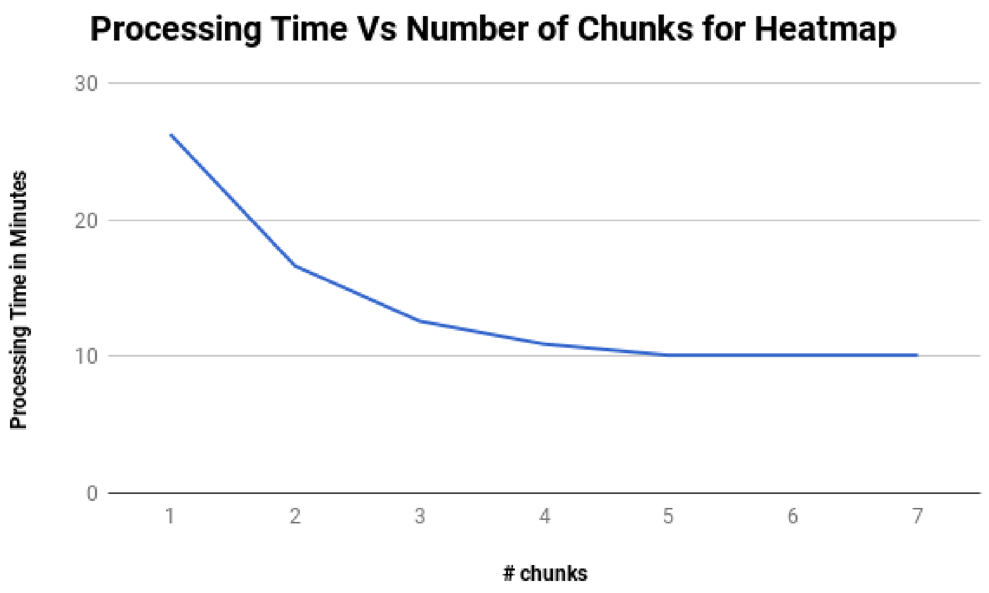


Figure 5.15: Total processing time of testing video for heatmaps.

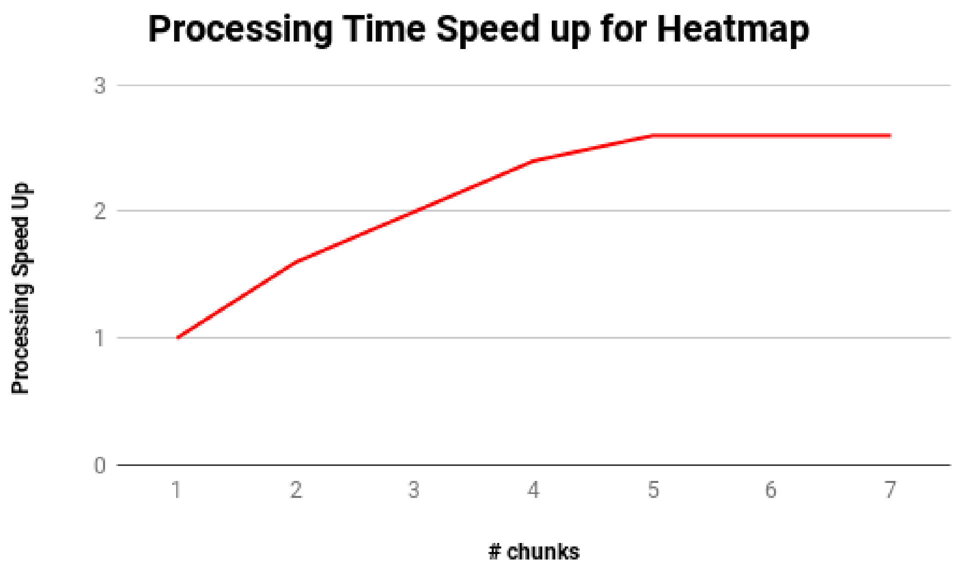


Figure 5.16: Processing time speed up of testing video for heatmaps.

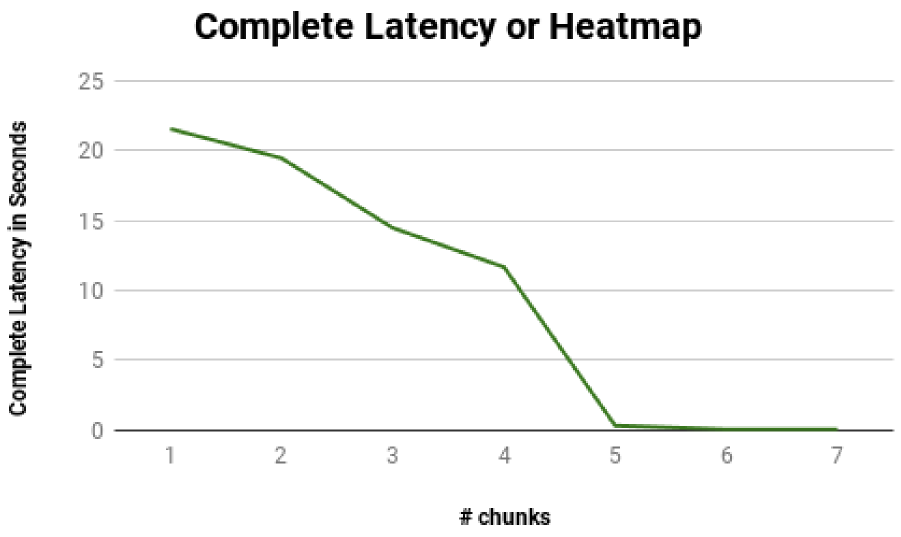


Figure 5.17: Complete latency of testing video for heatmaps.

5.3.5 Testing multiple video files

At the previous sections, the proposed DCB-Framework was tested for 1 hour testing video. In this section we selected the *Face Detection* use case for evaluating the DCB-Framework in processing multiple video files simultaneously in parallel. We considered the results of processing 1 video file sequentially and using 3, 4 and 5 chunks as a benchmark for evaluating the performance of processing from two to five 1 hour video files with same specifications. The processing time of multiple video files sequentially is calculated as multiple of processing 1 video file sequentially according to number of video files under testing. For example, the processing time of 1 video file sequentially equals 38.75 minutes. Therefore the processing time of 3 videos sequentially equals 38.75×3 minutes.

The processing evaluation metrics for multiple video files using face detection are listed in Table 5.7 and Table 5.8. From the collected results we found that when fixing number of chunks per each video, the total processing time decreases compared to processing the same number of video files in sequence. We achieved a higher speed up by increasing number of video files. This means that the extra resources needed to process multiple video files using DCB-Framework are reasonable compared to the processing gain we can achieve.

Table 5.7: Multiple files evaluation metrics.

# Videos	Processing Time in Minutes				Processing Speed up			Complete Latency		
	Sequential	3 chunks	4 chunks	5 chunks	3 chunks	4 chunks	5 chunks	3 chunks	4 chunks	5 chunks
1	38.75	17.33	15.30	13.28	2.2	2.5	2.9	18.9	17.02	14.9
2	77.50	23.00	20.58	17.82	3.4	3.8	4.3	12.7	10.90	9.8
3	116.25	26.98	24.12	19.32	4.3	4.8	6.0	9.9	8.30	7.1
4	155.00	31.33	30.37	22.18	5.0	5.1	6.7	7.9	6.60	5.6
5	193.75	34.35	32.66	24.20	5.6	5.9	8.0	6.5	5.70	4.6

Table 5.8: Multiple files FPS metrics.

# Videos	Spout avg. fps			Bolt task avg. fps		
	3 chunks	4 chunks	5 chunks	3 chunks	4 chunks	5 chunks
1	52.90	60.02	69.90	17.20	14.4	13.7
2	79.08	88.30	102.60	12.90	10.8	10.1
3	100.80	114.90	130.16	10.96	9.4	8.5
4	116.32	121.90	166.60	9.40	7.5	8.2
5	131.50	138.31	196.97	8.50	7.0	7.8

It is noted that there is a slight increase at total time of processing 1 or 2 or any number of files in parallel when fixing number of chunks. This occurs due to increasing number of bolt tasks to handle such high number of videos chunks which causes more context switching between Storm executors to utilize the available CPU cores besides the increasing communication between these bolt tasks and spout. Figure 5.18 shows that processing different number of video files in chunk mode requires almost same time of processing 1 video file with same number of chunks per individual video file with an added cumulative overhead. The speed up gain is increasing almost linearly with number of videos for the same number of chunks per video. The max processing speed up we recorded is 8x for processing 5 video files each splitted to 5 parallel chunks, see Figure 5.19. Figure 5.20 shows that the spout complete latency decrease as we increase number of video files and number of chunks as well.

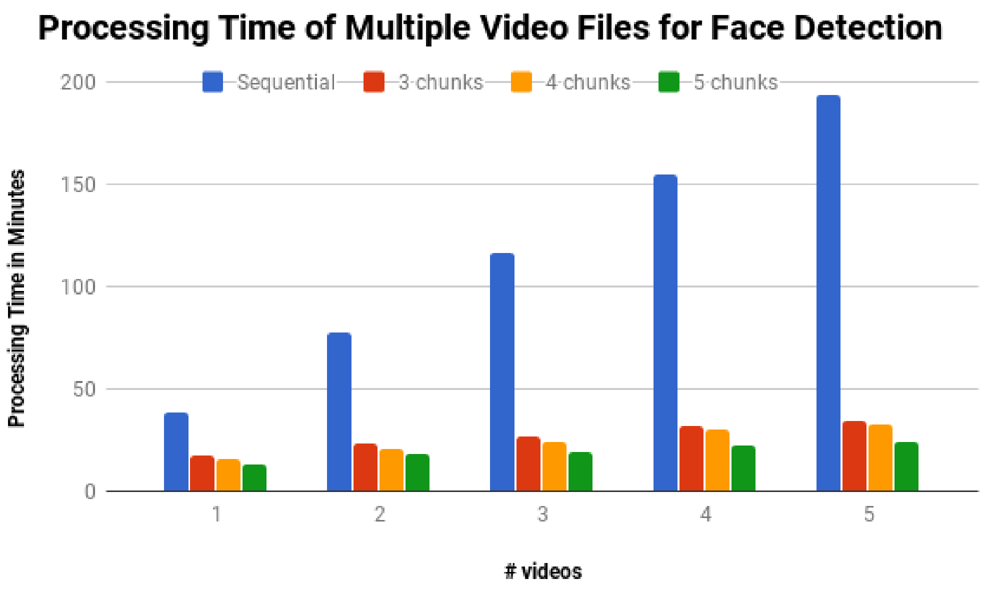


Figure 5.18: Total processing time for multiple files.

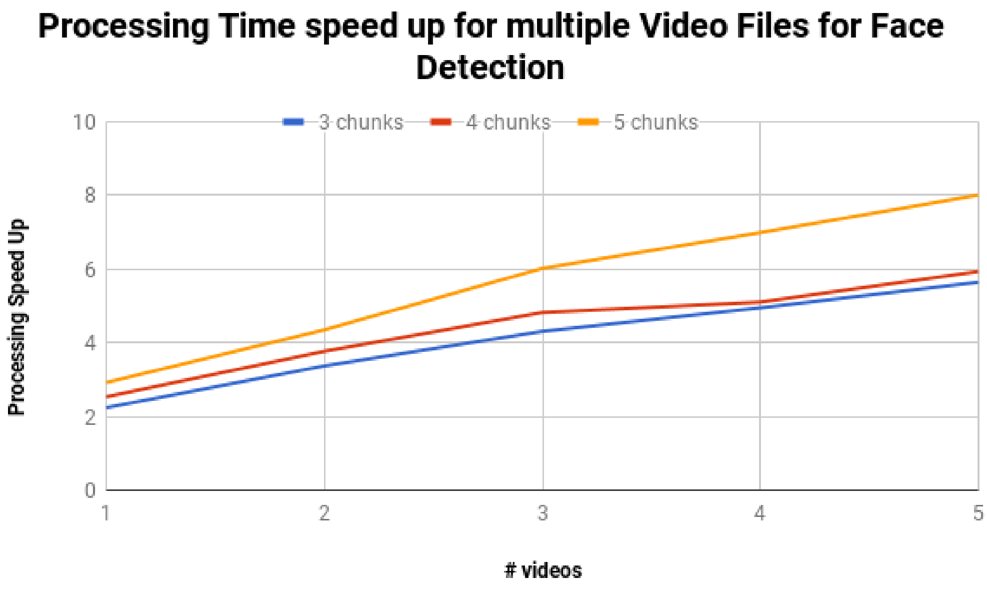


Figure 5.19: Processing time speed up for multiple files.

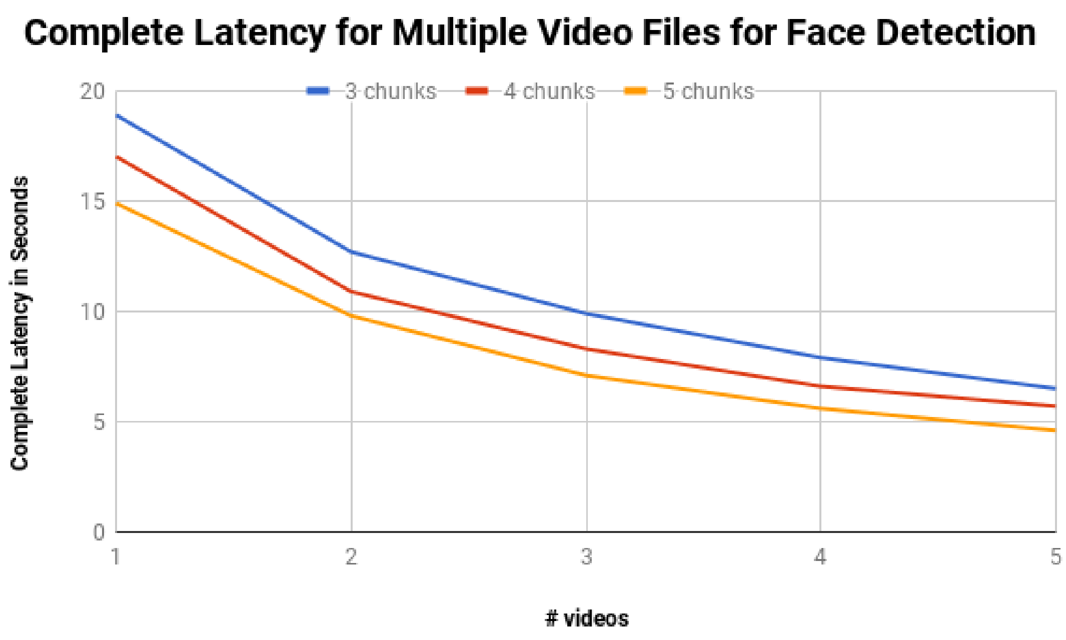


Figure 5.20: Complete latency for multiple files.

5.4 Results Observations

From previous evaluation testings we concluded to some important observations. First the speedup gain differs from one CV processing algorithm to another due to the following factors; the incoming input frame rate, size of individual frame, visual content of this frame and the processing logic of each algorithm. This explains why we got different speedup gain at the same number of chunks for each VP algorithm.

The complete latency of processing same amount of data decreases by increasing number of video chunks which means increasing number of consuming bolt tasks. This implies that one video frame waits less in the spout pending queue when there are much number of consuming bolt tasks that pull higher rates of data simultaneously. Decreasing complete latency is a vital advantage of DCB-Framework as it means we are almost processing CV sequential algorithms in real time with less overhead added to processing each input frame.

It is noticed that the spouts fps rates do not increase linearly with increasing number of bolt tasks because of the builtin Storm back pressure feature. In the beginning of processing, the spout fps starts from a high value and it keeps lowering as the bolt tasks receiving queues keep getting full. When the bolt tasks queues reach the high watermark or the spout pending queue reaches the spout max pending value, the spout pauses sending frames for processing a while until either of these situations is resolved. Reaching high watermark occurs usually in case of having large number of bolt tasks, thus the spout doesn't reach multiples of fps rates but it increases with acceptable rate. Different Storm configuration tuning could results in better processing rate.

The data received by Storm spout is distributed between all the available bolt tasks. That is why the total spout receiving frame fps is divided into underlying bolt tasks. So the individual bolt task processing fps decreases as we increase number of chunks for same input data however this low rate is multiplied by higher number of consumers. Spout and bolts fps are reflected in a higher processing throughput when the DCB-Framework is used.

For each CV algorithm we reached a certain saturation point which beyond it we do not get any extra gain nor speed up due to reaching fixed input rate. This happens according to several factors. The first one is the effect of spout back pressure that holds the spout from receiving or sending extra frames in case of topology congestion. The second one is that the communication channel between the DSFU and spout is governed by Apache Thrift data transmission rate. Thrift is a powerful cross-language communication tool, however it is not optimized for large size data similar to frame size that may reach 2 MByte. So, Thrift transmission rate reaches a certain upper value for each processing algorithm based on the algorithm processing speed itself. Despite these current limitations, our developed DCB-Framework reaches up to 8x of speed up and increases as we process much amount of video data which is the true essence of big-data processing.

5.5 Conclusion

In this chapter we performed a complete set of testing trials to evaluate the performance of the proposed DCB-Framework processing over a collection of sequential CV algorithms. The big-data processing framework used was Apache Storm with detailed illustration of topology architecture and configuration. We studied the effect of increasing number of chunks per 1 testing video file using Video Summarization, Face Detection, License Plate Recognition and Heatmaps. In all use cases, as we increase number of chunks, the DCB-Framework always outperforms the ordinary sequential processing by different speed up gains. Processing multiple video files in parallel also was studied under Face Detection algorithm. It also proved the practicality of the proposed processing algorithm and the higher gains it can achieve. The only drawback of the algorithm is the speed up gain saturation point that occurs when the VP algorithm speed hits Thrift transmission limits in addition to the effect of Storm back pressure feature.

Chapter 6: Conclusion and Future Work

6.1 Conclusion

As the technology evolves, a huge amount of video data are generated daily from enormous number of sources. Applying sequential CV algorithms over such huge amount of videos is not reliable using traditional big-data processing architecture.

In order to port CV algorithms with sequential data processing requirements in big-data processing frameworks we introduced in this thesis the DCB-Framework that is tailored for processing big-data videos within Apache Storm architecture. In our work we developed three new components that alter Storm data grouping mechanism and logic of handling sequential video frames while overcoming the challenges of processing sequential CV algorithms. We illustrated our new components: RCU, DSFU and DMRMU. We provided a conscious explanation of the working logic of each component and each one workflow.

The evaluation process of the introduced DCB-Framework was done using Apache Storm over testing video files. We first evaluated the performance of Video Summarization, Face Detection, License Plate Recognition and Heatmaps CV sequential algorithms in processing 1 hour video file for benchmarking. The purpose of testing different CV algorithms is to study the effect of changing input video's number of chunks versus the speed up gain we can get from each algorithm.

It was found that by using the proposed framework, we can get processing speed up ranges from 2.6x to 7.8x compared to processing the benchmarked video sequentially. The speedup gain varies depending on different working factors mentioned in the experimental testing analysis. For evaluating the proposed processing framework over multiple of concurrent input videos, we selected the Face Detection algorithm for such experiments. It was observed that by increasing number of input video files and number of chunk per video, the processing speedup increases significantly reaching x8 in case of processing 5 videos with 5 chunks each.

The evaluation process was done over 1 online powerful virtual machine, however the chunk-based framework is generic and can work on any number of machines within the processing cluster. The remarkable outcome of this framework is speeding sequential

CV algorithms, delivering accurate results in shorter time while utilizing the underlying processing resources in the most efficient way.

6.2 Future Work

The work of this thesis can be extended by working on several points:

1. Implementing a dynamic distributed chunk-based processing queue: The current implementation of the chunk-based processing is that the number of to-be processed chunks should equal number of available topology bolt tasks. At dynamic chunk queue concept, the initial chunk size provided before the topology submission is considered the final value without modification. This produces number of video chunks that may exceed available number of processing CPU cores, thus having not enough number of bolt tasks to process this number of chunks simultaneously. The dynamic chunk-queue allows as to process specific number of chunks at a time that equals the available number of cores and schedule the rest of chunks until the current ones finish processing. The DMRMU will assign one bolt task to one video chunk from first chunks batch as described before. When the processing of one video chunk is finished, the associated bolt task slot will be freed at assignment map in order to be assigned to another chunk from the upcoming pending batch.
2. Further investigation for saturation point: In order to extend the saturation point of each CV algorithm we need to fine tune storm configuration of back pressure feature. This tuning can give better spout and bolt fps rates that extend the processing enhancement saturation point to further point. Also, another study should be done to evaluate the effect of Thrift as a communication channel and if we can achieve better data rates if we replaced it by another tool. This would lead directly to an extended saturation point, thus higher processing rates at same number of chunks per video file.
3. Selecting optimal chunk size for multiple video files: For each computer algorithm, we need to formulate an equation that selects the optimal chunk size which gives the best performance for given number of video files taking into consideration the available resources.

References

[1] A. Labrinidis and H. V. Jagadish, "Challenges and opportunities with big data," *Proc. VLDB Endow.*, vol. 5, pp. 2032–2033, Aug. 2012.

[2] A. Gandomi and M. Haider, "Beyond the hype: Big data concepts, methods, and analytics," *International Journal of Information Management*, vol. 35, pp. 137–144, Apr. 2015.

[3] Statista: The Statistics Portal, "Hours of video uploaded to youtube every minute as of july 2015." Retrieved on May, 10, 2018 from: `https://www.statista.com/statistics/259477/hours-of-video-uploaded-to-youtube-every-minute/`.

[4] J. Manyika, M. Chui, B. Brown, J. Bughin, R. Dobbs, C. Roxburgh, et al., "Big data: The next frontier for innovation, competition, and productivity," tech. rep., McKinsey Global Institute, 2011.

[5] A. Woodie, "Analyzing video, the biggest data of them all," May 2016. Retrieved on May, 10, 2018 from: `https://www.datanami.com/2016/05/26/analyzing-video-biggest-data/`.

[6] D. Laney, "3-d data management: Controlling data volume, velocity, and variety," *Application Delivery Strategies by META Group Inc*, vol. 6, p. 949, 01 2001.

[7] S. Mukherjee and R. Shaw, "Big data concepts, applications, challenges and future scope," *International Journal of Advanced Research in Computer and Communication Engineering*, vol. 5, Feb. 2016.

[8] R. Sint, S. Schaffert, S. Stroka, and R. Ferstl, "Combining unstructured, fully structured and semi-structured information in semantic wikis," in *Proceedings of the 4th Semantic Wiki Workshop (SemWiki) at the 6th European Semantic Web Conference, ESWC*, 2009.

[9] M. Rasool and W. Khan, "Big data: Study in structured and unstructured data," *HCTL Open International Journal of Technology Innovations and Research (IJTIR)*, vol. 14, Apr. 2015.

[10] M. v. Rijmenam, "Blog post: Why the 3v's are not sufficient to describe big data." Retrieved on March, 13, 2018 from: `https://datafloq.com/read/3vs-sufficient-describe-big-data/166`.

[11] S. Shukla, V. Kukade, and S. Mujawar, "Big data: Concept, handling and challenges: An overview," *International Journal of Computer Applications*, vol. 114, Mar. 2015.

[12] J. Kelly, "Big data vendor revenue and market forecast 2013-2017." Retrieved on March 17, 2018 from: `http://wikibon.org/wiki/v/Big_Data_Vendor_Revenue_and_Market_Forecast_2013-2017#Wikibon.E2.80.99s_Big_Data_Market_Forecast`.

[13] S. Kaur and S. Cheema, "Review paper on big data: Applications and different tools," *International Journal of Advanced Research in Computer and Communication Engineering*, vol. 6, Jun. 2017.

[14] S. Soundararajan, "How social media companies use big data." Retrieved on March, 17, 2018 from: `https://datafloq.com/read/how-social-media-companies-use-big-data/1957`.

[15] N. Elgendy and A. Elragal, "Big data analytics: A literature review paper," 08 2014. Lecture Notes in Computer Science.

[16] Z. Zheng, P. Wang, J. Liu, and S. Sun, "Real-time big data processing framework: Challenges and solutions," *Applied Mathematics Information Sciences, An International Journal*, no. 6, pp. 3169–3190, 2015.

[17] W. Inoubli, S. Aridhi, H. Mezni, and A. Jung, "Big data frameworks: A comparative study," *CoRR: Computing Research Repository*, vol. abs/1610.09962, 2016.

[18] Apache Software Foundation, "Apache hadoop official website." Retrieved on March, 21, 2018 from: `http://hadoop.apache.org/`.

[19] K. Subhash Raste, "BIG DATA ANALYTICS HADOOP PERFORMANCE ANALYSIS," Master's thesis, Faculty of San Diego State University, 2014.

[20] K. Shvachko, H. Kuang, S. Radia, and R. Chansler, "The Hadoop Distributed File System," in *Mass Storage Systems and Technologies (MSST), 2010 IEEE 26th Symposium on*, pp. 1–10, IEEE, 2010.

[21] J. Dean and S. Ghemawat, "Mapreduce: Simplified data processing on large clusters," in *Proceedings of the 6th Conference on Symposium on Opearting Systems Design & Implementation - Volume 6*, OSDI'04, (Berkeley, CA, USA), pp. 10–10, USENIX Association, 2004.

[22] V. K. Vavilapalli, A. C. Murthy, C. Douglas, S. Agarwal, M. Konar, R. Evans, T. Graves, J. Lowe, H. Shah, S. Seth, B. Saha, C. Curino, O. O'Malley, S. Radia, B. Reed, and E. Baldeschwieler, "Apache hadoop yarn: Yet another resource negotiator," in *Proceedings of the 4th Annual Symposium on Cloud Computing*, SOCC '13, (New York, NY, USA), pp. 5:1–5:16, ACM, 2013.

[23] S. Saggar1, N. Khurana, and R. Saggar, "Hadoop multiple job trackers with master slave replication model," *International Journal of Advance Research in Computer Science and Management Studies*, vol. 3, pp. 135–144, Aug. 2015.

[24] P. por Cristian, "Hadooper blogspot: A brief view to the platform." Retrieved on March, 26, 2018 from: `http://hadooper.blogspot.com.eg/2010/10/brief-view-to-platform.html`.

[25] A. Kulkarni and M. Khandewal, "Survey on hadoop and introduction to yarn," *International Journal of Emerging Technology and Advanced Engineering*, vol. 4, May. 2014.

[26] A. Murthy, V. Vavilapalli, D. Eadline, J. Niemiec, and J. Markham, *Apache Hadoop YARN: Moving beyond MapReduce and Batch Processing with Apache Hadoop 2*. Addison-Wesley Professional., 2014.

[27] M. Usama, M. Liu, and M. Chen, "Job schedulers for big data processing in hadoop environment: testing real-life schedulers using benchmark programs," *Digital Communications and Networks*, vol. 3, no. 4, pp. 260 – 273, 2017. Big Data Security and Privacy.

[28] A. Bhadani and D. Jothimani, "Big data: Challenges, opportunities and realities," *CoRR: Computing Research Repository*, vol. abs/1705.04928, 2017.

[29] Apache Software Foundation, "Apache spark official website." Retrieved on March, 28, 2018 from: `https://spark.apache.org/`.

[30] V. S. Jonnalagadda, P. Srikanth, K. Thumati, and S. H. Nallamala, "A review study of apache spark in big data processing," *International Journal of Computer Science Trends and Technology (IJCST)*, vol. 4, no. 3, pp. 93–98, 2016.

[31] M. Zaharia, M. Chowdhury, M. J. Franklin, S. Shenker, and I. Stoica, "Spark: Cluster computing with working sets," in *Proceedings of the 2Nd USENIX Conference on Hot Topics in Cloud Computing*, HotCloud'10, (Berkeley, CA, USA), pp. 10–10, USENIX Association, 2010.

[32] D. García-Gil, S. Ramírez-Gallego, S. García, and F. Herrera, "A comparison on scalability for batch big data processing on apache spark and apache flink," *Big Data Analytics*, vol. 2, p. 1, Mar 2017.

[33] M. Zaharia, R. Xin, P. Wendell, T. Das, M. Armbrust, A. Dave, X. Meng, J. Rosen, S. Venkataraman, M. J. Franklin, A. Ghodsi, J. Gonzalez, S. Shenker, and I. Stoica, "Apache spark: a unified engine for big data processing," *Commun. ACM*, vol. 59, pp. 56–65, 2016.

[34] A. Grishchenko, "Apache spark architecture," Nov 2015. Retrieved on March, 31, 2018 from: `https://www.slideshare.net/AGrishchenko/apache-spark-architecture`.

[35] J. Laskowski, "Online book: Mastering apache spark (2.3.0)," 2018. Retrieved on March, 31, 2018 from: `https://www.gitbook.com/book/jaceklaskowski/mastering-apache-spark/details`.

[36] Apache Software Foundation, "Apache mesos official website." Retrieved on April, 2, 2018 from: `http://mesos.apache.org/`.

[37] Apache Software Foundation, "Job scheduling in spark." Retrieved on April, 2, 2018 from: `https://spark.apache.org/docs/latest/job-scheduling.html`.

[38] Data Flair, "Limitations of apache spark ways to overcome spark drawbacks," Apr 2017. Retrieved on April, 2, 2018 from: `https://data-flair.training/blogs/limitations-of-apache-spark/`.

[39] A. Ghaffar Shoro and T. Soomro, "Big data analysis: Apache spark perspective," in *Global Journal of Computer Science and Technology*, vol. 15, Jan 2015.

[40] Apache Software Foundation, "Apache storm official website." Retrieved on October, 1, 2017 from: `http://storm.apache.org/`.

[41] MapR, "Apache storm on hadoop." Retrieved on October, 1, 2017 from: `https://www.mapr.com/products/product-overview/apache-storm-hadoop`.

[42] A. Toshniwal, S. Taneja, A. Shukla, K. Ramasamy, J. M. Patel, S. Kulkarni, J. Jackson, K. Gade, M. Fu, J. Donham, N. Bhagat, S. Mittal, and D. Ryaboy,

"Storm@twitter," in *Proceedings of the 2014 ACM SIGMOD International Conference on Management of Data*, SIGMOD '14, (New York, NY, USA), pp. 147–156, ACM, 2014.

[43] C. Prakash, "Apache storm : Architecture overview," Jan 2016. Retrieved on March, 20, 2018 from: `https://www.linkedin.com/pulse/apache-storm-architecture-overview-chandan-prakash`.

[44] J. S. v. d. Veen, B. v. d. Waaij, E. Lazovik, W. Wijbrandi, and R. J. Meijer, "Dynamically scaling apache storm for the analysis of streaming data," in *2015 IEEE First International Conference on Big Data Computing Service and Applications*, pp. 154–161, March 2015.

[45] L. Aniello, R. Baldoni, and L. Querzoni, "Adaptive online scheduling in storm," in *Proceedings of the 7th ACM International Conference on Distributed Event-based Systems*, DEBS '13, (New York, NY, USA), pp. 207–218, ACM, 2013.

[46] S. Surshanov, "Using apache storm for big data," *COMPUTER MODELLING NEW TECHNOLOGIES, cmnt*, pp. 14–17, 2015.

[47] F. Jiang, "Exoblog: Enabling site-aware scheduling for apache storm in exogeni," Apr 2015. Retrieved on March, 20, 2018 from: `http://www.exogeni.net/2015/04/enabling-site-aware-scheduling-for-apache-storm-in-exogeni/`.

[48] K. Ching Li, H. Jiang, L. T. Yang, and A. Cuzzocrea, *Big Data: Algorithms, Analytics, and Applications*. CRC Press, 2015.

[49] B. Peng, M. Hosseini, Z. Hong, R. Farivar, and R. Campbell, "R-storm: Resource-aware scheduling in storm," Dec 2015.

[50] "Metadata-aware scheduler for apache storm," Apr 2015. Retrieved on March, 26, 2018 from: `https://dcvan24.wordpress.com/2015/04/07/metadata-aware-custom-scheduler-in-storm/`.

[51] V. Cardellini, V. Grassi, F. L. Presti, and M. Nardelli, "On qos-aware scheduling of data stream applications over fog computing infrastructures," in *2015 IEEE Symposium on Computers and Communication (ISCC)*, pp. 271–276, July 2015.

[52] A. Jain, *Mastering Apache Storm: Processing big data streams in real time*. Birmingham: Packt Publishing, 2017.

[53] "Wikipedia: Computer vision," Apr 2018. Retrieved on April, 15, 2018 from: `https://en.wikipedia.org/wiki/Computer_vision`.

[54] J. Fernandez, “Survey on computer vision state-of-the-art and applications for big data,” Jun 2015. Retrieved on April, 16, 2018 from: https://www.slideshare.net/javierfdr/computer-vision-state-of-the-art?from_action=save.

[55] M. Thompson, “Electronic design blog: Conquer the challenge of integrating efficient embedded vision,” May 2015. Retrieved on April, 15, 2018 from: http://www.electronicdesign.com/embedded/conquer-challenge-integrating-efficient-embedded-vision.

[56] J. van de Loosdrecht, “Accelerating sequential computer vision algorithms using commodity parallel hardware,” Master’s thesis, NHL University, 09 2013.

[57] “OpenMP official website.” Retrieved on May, 11, 2018 from: http://www.openmp.org/.

[58] V. Chaudhary and J. K. Aggarwal, *Parallelism in Computer Vision: a Review*, pp. 271–309. New York, NY: Springer New York, 1990.

[59] S. Kadam, “Parallelization of low-level computer vision algorithms on clusters,” in *2008 Second Asia International Conference on Modelling x00026; Simulation (AMS)*, pp. 113–118, May 2008.

[60] T. Deneke, J. Lilius, and S. Lafond, “Scalable distributed video transcoding architecture,” 2012.

[61] F. Lao, X. Zhang, and Z. Guo, “Parallelizing video transcoding using map-reduce-based cloud computing,” 05 2012.

[62] C. V. FANG, “Large-Scale Video Analytics on Hadoop,” Aug 2013. Retrieved on May, 12, 2018 from: https://content.pivotal.io/blog/large-scale-video-analytics-on-hadoop.

[63] “Pivotal Official Website.” Retrieved on May, 12, 2018 from: https://pivotal.io/.

[64] Z. Lin, L. Zhen, C. Yingmei, and T. Yuqin, “The video monitoring system based on big data processing,” in *2014 7th International Conference on Intelligent Computation Technology and Automation*, pp. 865–868, Oct 2014.

[65] R. H. M. Bhuiyan, “Low delay video transcoding services on distrubuted computing platform.,” Master’s thesis, , Department of Applied Signal Processing Ericsson AB. bo Akademi University., 2016.

[66] "StormCV," Oct 2016. Retrieved on May, 12, 2018 from: `https://github.com/sensorstorm/StormCV`.

[67] S. Dutt and A. Kalra, "A scalable and robust framework for intelligent real-time video surveillance," in *2016 International Conference on Advances in Computing, Communications and Informatics (ICACCI)*, pp. 212–215, Sept 2016.

[68] Apache Software Foundation, "Apache Thrift Official Website." Retrieved on May, 1, 2018 from: `https://thrift.apache.org/`.

[69] A. Agarwal, M. Slee, and M. Kwiatkowski, "Thrift: Scalable cross-language services implementation," tech. rep., Facebook, 4 2007.

[70] Wikipedia, "Video compression picture types." Retrieved on May, 2, 2018 from: `https://en.wikipedia.org/wiki/Video_compression_picture_types`.

[71] "White paper: An explanation of video compression techniques," tech. rep., Axis Communications, 2008.

[72] Wikipedia, "Group of pictures." Retrieved on May, 2, 2018 from: `https://en.wikipedia.org/wiki/Group_of_pictures`.

[73] Redis Labs, "Redis official website." Retrieved on May, 2, 2018 from: `https://redis.io/`.

[74] "White paper: Database caching strategies using redis," tech. rep., Amazon Web Services, May 2017.

[75] M. Paksula, "Persisting objects in redis key-value database," 2010. University of Helsinki, Department of Computer Science.

[76] "lettuce - advanced java redis client." Retrieved on May, 2, 2018 from: `https://github.com/lettuce-io/lettuce-core`.

[77] "lua - the programming language-: Official website." Retrieved on May, 2, 2018 from: `http://www.lua.org/`.

[78] HORTONWORKS, "Apache Storm Component Guide: Configuring Storm Resource Usage." Retrieved on May, 5, 2018 from: `https://docs.hortonworks.com/HDPDocuments/HDP2/HDP-2.6.4/bk_storm-component-guide/content/config-storm-settings.html`.

[79] "ONE2TEAM Carrers Blog: HOW TO TUNE APACHE STORMS AUTOMATIC BACK PRESSURE?," Jun. 2016. Retrieved on May, 5, 2018 from: https://jobs.one2team.com/apache-storms/.

[80] P. T. Goetz, "Storm 1.0.0 released," Apr. 2016. Retrieved on May, 5, 2018 from: http://storm.apache.org/2016/04/12/storm100-released.html.

[81] NEC Corporation, "NEC's Video Face Recognition Technology Ranks First in NIST Testing," Mar. 2017. Retrieved on May, 24, 2018 from: https://uk.nec.com/en_GB/press/201703/global_20170316_01.html.

استخدمنا العديد من خوارزميات رؤية الحاسب المتسلسلة بما في ذلك كشف الوجه ،تلخيص الفيديو ،التعرف على لوحة ترخيص المركبات و خرائط الحرارة لتقييم العمل المقترح. تم دمج هذه الخوارزميات في منصة الاختبار الخاصة بنا. حقق إطار المعالجة المتوازي القائم على تجزئ البيانات تقدم فى سرعة المعالجة بعامل تسريع من 2.6 إلى 7.8 على أساس خوارزمية رؤية الحاسب المستخدمة. كما تم تقييم معالجة ملفات الفيديو المتعددة ضد عدد مختلف من أجزاء الفيديو باستخدام خوارزمية الكشف عن الوجه ليصل معامل تسريع المعالجة في هذه الحالة إلى 8 أضعاف المعالجة المتسلسلة العادية.

ملخص الرسالة

يتناول البحث المقدم في هذه الأطروحة فكرة موازاة خوارزميات رؤية الحاسب المتسلسلة، مثل اكتشاف الحركة، تتبع الأشياء ، إلخ ، على أدوات البيانات الضخمة. خوارزميات رؤية الحاسب المتسلسلة لديها قيود على كيفية معالجة إطارات مقاطع الفيديو . في هذه الخوارزميات تكون العلاقة بين الإطارات المتتابعة جزءاً مهماً من تسلسل عمل الخوارزمية نتيجة لأن ناتج معالجة إطار فيديو واحد يعتمد مباشرا على ناتج معالجة الإطار أو الإطارات السابقة.

تقوم معظم أطر معالجة البيانات الضخمة الحالية بتوزيع بيانات الإدخال بشكل عشوائي عبر وحدات المعالجة المتاحة للاستفادة منها بكفاءة و للحفاظ على عدالة توزيع الأحمال. وبالتالي ،فإن أطر البيانات الضخمة الحالية ليست مناسب لمعالجة بيانات الفيديو ذات الاعتماد المتبادل بين الإطارات. عند معالجة هذه الخوارزميات المتسلسلة على أدوات البيانات الضخمة ، لن يؤدي تقسيم إطارات الفيديو و توزيعها على وحدات المعالجة المتوفرة إلى النتائج الصحيحة. وبالتالي ، فإن ميزة خوارزميات المعالجة التتابعية في إطار البيانات الضخمة تصبح محدودة فقط في حالات معينة حيث تأتي تيارات الفيديو من مصادر دخل مختلف.

في هذه الأطروحة ، نقدم إطاراً كاملاً يُمكّن أدوات البيانات الضخمة من تنفيذ خوارزميات رؤية الحاسب المتسلسلة بطريقة قابلة للتوسع والتوازي مع تعديلات محدودة. هدفنا الرئيسي هو موازاة عملية المعالجة من أجل تسريع وقت المعالجة المطلوبة. الفكرة الرئيسية هي تقسيم ملفات الفيديو الكبيرة المدخلة إلى أجزاء صغيرة يمكن معالجتها بالتوازي دون التأثير على جودة المخرجات الناتجة. لقد قمنا بتطوير تقنية تجميع بيانات ذكية تقوم بتوزيع قطع البيانات بين موارد المعالجة المتوفرة وجمع النتائج من كل جزء. تم أيضا استخدام مقطع بيانات قائم على تقطيع أجزاء بيانات الإدخال بشكل متزامن ليتم معالجتها بالتوازي. ثم تقوم خوارزمية التجميع لدينا بالتأكد من أن جميع الإطارات التي تنتمي إلى نفس جزء الفيديو المقطع يتم توزيعها إلى مراكز المعالجة الخاصة بها بشكل صحيح.

لتقييم أداء إطار العمل المقترح المبنى على أساس تجزيء البيانات ، أجرينا العديد من الاختبارات التجريبية باستخدام إطار البيانات الضخمة المعروف (ستورم) للاستفادة من قدرته على المعالجة في الوقت الفعلي. تم تعديل طريقة عمل (ستورم) ليدعم تقسيم إطارات الفيديو المدخلة و المعالجة المتوازية معا. لقد تم دراسة إطار العمل المقترح لدينا ضد عدد مختلف من الأجزاء المقطعة على مدى فيديوهات الاختبار مدتها ساعة واحدة.

مهندس:	نورهان مجدى سيد عثمان
تاريخ الميلاد:	1990\5\27
الجنسية:	مصرى
تاريخ التسجيل:	2014\3\1
تاريخ المنح:	2018
القسم:	هندسة الإلكترونيات و الإتصالات الكهربية
الدرجة:	ماجستير العلوم

المشرفون:

أ.د. حسام على حسن فهمى
د. محمد محمد ريحان
المدير التقنى بشركة أفيدبيم

الممتحنون:

أ.د حسام على حسن فهمى	(المشرف الرئيسي)
د. محمد محمد ريحان	(المشرف)
المدير التقنى بشركة أفيدبيم	
أ.د السيد عيسى عبده حميد	(الممتحن الداخلي)
أ.د خالد مصطفى السيد	(الممتحن الخارجي)
كلية الحاسبات و المعلومات جامعة القاهرة	

عنوان الرسالة:

موازاة خوارزميات رؤية الحاسب المتسلسلة على البيانات الضخمة باستخدام إطار توزيع أجزاء البيانات

الكلمات الدالة:

البيانات الكبيرة، رؤية الحاسب الآلي, الحوسبة المتوازية ، معالجة مقاطع الفيديو, أجزاء الفيديو

ملخص الرسالة:

في هذه الأطروحة ، نقدم إطارا عمل كامل يمكّن أطر البيانات الكبيرة من تشغيل خوارزميات رؤية الحاسب المتسلسلة بطريقة قابلة للتوسع والتوازي. فكرتنا هو تقسيم ملفات الفيديو المدخلة إلى أجزاء صغيرة يمكن معالجتها بالتوازي دون التأثير على جودة الناتج . لتحقيق هذا الهدف قمنا بتطوير تقنية تجميع البيانات الذكية التي توزع أجزاء الفيديو المقطعة بين موارد المعالجة المتاحة وجمع النتائج من كل قطعة أسرع من استخدام طرق المعالجة المتسلسلة المستقلة

موازاة خوارزميات رؤية الحاسب المتسلسلة على البيانات الضخمة باستخدام إطار توزيع أجزاء البيانات

إعداد

نورهان مجدى سيد عثمان

رسالة مقدمة إلى كلية الهندسة – جامعة القاهرة
كجزء من متطلبات الحصول على درجة
ماجستير العلوم
في
هندسة الإلكترونيات و الإتصالات الكهربية

يعتمد من لجنة الممتحنين:

الاستاذ الدكتور: حسام على حسن فهمى	المشرف الرئيسى
الدكتور: محمد محمد ريحان المدير التقنى بشركة أفيدبيم	المشرف
الاستاذ الدكتور: السيد عيسى عبده حميد	الممتحن الداخلي
الاستاذ الدكتور: خالد مصطفى السيد كلية الحاسبات و المعلومات, جامعة القاهرة	الممتحن الخارجى

كليـة الهندسـة - جامعـة القاهـرة
الجيـزة - جمهوريـة مصـر العربيـة
2018

موازاة خوارزميات رؤية الحاسب المتسلسلة على البيانات الضخمة باستخدام إطار توزيع أجزاء البيانات

إعداد

نورهان مجدى سيد عثمان

رسالة مقدمة إلى كلية الهندسة – جامعة القاهرة
كجزء من متطلبات الحصول على درجة
ماجستيرالعلوم
في
هندسة الإلكترونيات و الإتصالات الكهربية

تحت اشراف

ا.د. حسام على حسن فهمى	**د. محمد محمد ريحان**
أستاذ بقسم هندسة الإلكترونيات والإتصالات، كلية الهندسة، جامعة القاهرة	المدير التقني التنفيذي بشركة افيدبيم

كليــة الهندســـة - جامعــة القاهــرة
الجيـزة - جمهوريـة مصـر العربيــة

2018

موازاة خوارزميات رؤية الحاسب المتسلسلة على البيانات الضخمة باستخدام إطار توزيع أجزاء البيانات

إعداد

نورهان مجدى سيد عثمان

رسالة مقدمة إلى كلية الهندسة – جامعة القاهرة
كجزء من متطلبات الحصول على درجة
ماجستير العلوم
في
هندسة الإلكترونيات و الإتصالات الكهربية

كليــة الهندســة - جامعــة القاهــرة
الجيـزة - جمهوريـة مصـر العربيــة

2018