Performance Evaluation of Job Scheduling and Resource Allocation in Apache Spark

A Thesis Submitted to the
College of Graduate and Postdoctoral Studies
in Partial Fulfillment of the Requirements
for the degree of Master of Science
in the Department of Computer Science
University of Saskatchewan
Saskatoon

By
Habib Ado Sabiu

Permission to Use

In presenting this thesis in partial fulfilment of the requirements for a Postgraduate degree from the University of Saskatchewan, I agree that the Libraries of this University may make it freely available for inspection. I further agree that permission for copying of this thesis in any manner, in whole or in part, for scholarly purposes may be granted by the professor or professors who supervised my thesis work or, in their absence, by the Head of the Department or the Dean of the College in which my thesis work was done. It is understood that any copying or publication or use of this thesis or parts thereof for financial gain shall not be allowed without my written permission. It is also understood that due recognition shall be given to me and to the University of Saskatchewan in any scholarly use which may be made of any material in my thesis.

Requests for permission to copy or to make other use of material in this thesis in whole or part should be addressed to:

Head of the Department of Computer Science
University of Saskatchewan
176 Thorvaldson Building, 110 Science Place
Saskatoon, Saskatchewan S7N 5C9
Canada

OR

Dean
College of Graduate and Postdoctoral Studies
University of Saskatchewan
116 Thorvaldson Building, 110 Science Place
Saskatoon, Saskatchewan S7N 5C9
Canada

Abstract

Advancements in data acquisition techniques and devices are revolutionizing the way image data are collected, managed and processed. Devices such as time-lapse cameras and multispectral cameras generate large amount of image data daily. Therefore, there is a clear need for many organizations and researchers to deal with large volume of image data efficiently. On the other hand, Big Data processing on distributed systems such as Apache Spark are gaining popularity in recent years. Apache Spark is a widely used in-memory framework for distributed processing of large datasets on a cluster of inexpensive computers. This thesis proposes using Spark for distributed processing of large amount of image data in a time efficient manner. However, to share cluster resources efficiently, multiple image processing applications submitted to the cluster must be appropriately scheduled by Spark cluster managers to take advantage of all the compute power and storage capacity of the cluster. Spark can run on three cluster managers including Standalone, Mesos and YARN, and provides several configuration parameters that control how resources are allocated and scheduled. Using default settings for these multiple parameters is not enough to efficiently share cluster resources between multiple applications running concurrently. This leads to performance issues and resource underutilization because cluster administrators and users do not know which Spark cluster manager is the right fit for their applications and how the scheduling behaviour and parameter settings of these cluster managers affect the performance of their applications in terms of resource utilization and response times.

This thesis parallelized a set of heterogeneous image processing applications including Image Registration, Flower Counter and Image Clustering, and presents extensive comparisons and analyses of running these applications on a large server and a Spark cluster using three different cluster managers for resource allocation, including Standalone, Apache Mesos and Hodoop YARN. In addition, the thesis examined the two different job scheduling and resource allocations modes available in Spark: static and dynamic allocation. Furthermore, the thesis explored the various configurations available on both modes that control speculative execution of tasks, resource size and the number of parallel tasks per job, and explained their impact on image processing applications. The thesis aims to show that using optimal values for these parameters reduces jobs makespan, maximizes cluster utilization, and ensures each application is allocated a fair share of cluster resources in a timely manner.

Acknowledgements

I would like to express my sincere appreciation to everyone who helped make this work possible. My utmost gratitude goes to my incredibly patient supervisors; Dr. Dwight Makaroff and Dr. Winfried Grassmann, who helped me immensely throughout my Masters program at the University of Saskatchewan. Thank you for providing me with endless support, guidance, and your valuable time. I would also like to thank my perpetually tolerant committee members: Dr. Derek Eager, Dr. Kevin Schneider and Dr. Seok-Bum Ko, for their insightful comments and constructive suggestions. Finally, I would like to thank my parent and siblings for their continuous encouragement and support.

This thesis is gratefully dedicated to the memory of my beloved father, Ado Sabiu and to my lovely mother, Hadiza Mukhtar Sarkinbai, who has always been my role model and inspiration. Thank you for all the love and support.

- Habib

Contents

List of Tables

List of Figures

List of Abbreviations

API	Application Programming Interface
DRF	Dominant Resource Fairness
FIFO	First In First Out
HDFS	Hadoop Distributed File System
HPC	High Performance Computing
JVM	Java Virtual Machine
MPI	Message Passing Interface
NDVI	Normalized Difference Vegetation Index
RAM	Random Access Memory
RDD	Resilient Distributed Datasets
SLA	Service Level Agreement
SLO	Service Level Objectives
SQL	Structured Query Language
YARN	Yet Another Resource Negotiator

Chapter 1

Introduction

Big Data frameworks such as Hadoop [62] and Spark [69] are used for distributed processing of large datasets that scale to thousands of processors and terabytes of Random Access Memory (RAM) in cluster configurations. Hadoop is the open source implementation of the MapReduce framework [14] developed by Google in 2004. MapReduce provides a strategy for implementing and executing large scale data intensive applications in parallel on clusters. This parallel execution of applications reduces the latency between the submission of jobs and the availability of results, helping in batch processing of large datasets. Spark [69] was introduced in 2010 to support certain classes of applications that are not suitable for MapReduce such as iterative machine learning algorithms and interactive requests. The primary advantage of Spark over other data processing engines such as Hadoop is the use of distributed memory abstraction called Resilient Distributed Datasets (RDD), which provides in-memory computation and supports coarse-grained transformations to improve performance of interactive request and iterative algorithms. In-memory is a concept used to store massive amounts of data in the main RAM of computers across a cluster to provide fast access to the data and minimize the requirements of disk I/O.

Modern image data acquisition devices such timelapse cameras and drones equipped with multispectral cameras generate large amounts of image data used in sophisticated image processing, statistical, and machine learning algorithms to extract useful information for crop breeders and producers. This thesis proposes using Spark as a viable means of distributing and running multiple image processing applications concurrently on a cluster to minimize jobs makespan (that is the total time that elapses from submitting the first job until the end of the job that finishes last). In order to share cluster resources efficiently, the multiple jobs submitted to the cluster must be appropriately scheduled by the cluster manager to take advantage of all the compute power, memory and networking capacity of the cluster. Spark relies on three cluster managers including a) Standalone which comes as part of the Spark distribution, b) Mesos, and c) YARN for job scheduling and resource allocation. In addition to supporting three different cluster managers, Spark provides two modes of resource allocation: static allocation and dynamic allocation; as well, many configuration parameters are available that control speculative execution of tasks and size and number of resources to allocate to the submitted applications.

The default mode for resource allocation available on the three Spark supported cluster managers considered in this thesis is *Static Resource Allocation*. With this approach, applications are assigned resources at

the start of execution, and hold on to them for their whole duration. Although this approach could provide speedup gains in a single user or a batch oriented cluster, it is not ideal in a shared cluster where multiple applications need to run concurrently, since applications that are not able to utilize all their allocated resources may still hold on to them. This increases the makespan of the set of applications and leads to inefficient utilization of cluster resources.

Initial experiments that were performed showed that the default parameter settings for this approach allow applications to grab all the cluster resources when running on Standalone and Mesos cluster managers, or request very minimal number of executors regardless of available resources when running on YARN cluster manager. Since this is not efficient for sharing cluster resources between multiple applications, Spark introduced *Dynamic Resource Allocation*, which is specifically designed to dynamically adjust the resources that applications use based on the current workload. With *Dynamic Resource Allocation*, applications can request additional resources when they need them, and give these resources back to the cluster manager when they are no longer used, leading to better cluster utilization and efficient resource sharing between multiple applications.

There are also many configuration parameters that control the behaviour of this resource allocation mode such as the initial, minimum and maximum number of executors that could be allocated to applications, when to remove executors that have been idle for more than a certain duration of time and the duration for which pending tasks must wait before an application can request a new executor. The default settings for these parameters are not sufficient, however, to share resources efficiently between multiple applications running concurrently on a shared cluster. Finally, different cluster managers have different job scheduling and resource allocation policies, which could affect the performance of image processing applications.

Cluster administrators must decide which cluster manager is the best fit for their specific type of applications, what mode of resource allocation to use, and how to tune the multiple scheduling and resource allocation parameters of Spark to get the best application performance and efficient resource utilization. Therefore, it is very important to understand if and how the different resource allocation policies implemented in the three Spark cluster managers affect performance. This thesis analyzes and evaluates the performance impact of the two different job scheduling and resource allocation modes available on Spark, including static and dynamic allocation, and explore the various Spark configurations that control speculative execution of tasks, resource size and the number of parallel tasks per job, and their effects on image processing applications.

1.1 Big Data Framework Concepts

MapReduce [14] is a programming model for distributed processing of large datasets on clusters of commodity hardware (relatively inexpensive and widely available), where nodes in the cluster are equipped with both storage and computation resources. The MapReduce algorithm is made up of two phases, namely Map and Reduce. The Map phase takes a dataset of key/value pairs, applies a user defined function to all the elements

in the dataset and generates a set of intermediate key/value pairs as output. The Reduce phase aggregates the intermediate key/value pairs from Map output and merge all the values with the same intermediate key. Figure 1.1 shows the various stages of MapReduce. Since many real world data processing tasks can be modelled in this way [14], MapReduce has been widely adopted in both academia and industry for batch processing of large datasets [56]. The advantages of MapReduce are that it provides a simple model that supports automatic parallelization, fault-tolerance, data distribution and load balancing. It is also easy to scale big data processing over multiple computing nodes using this model.

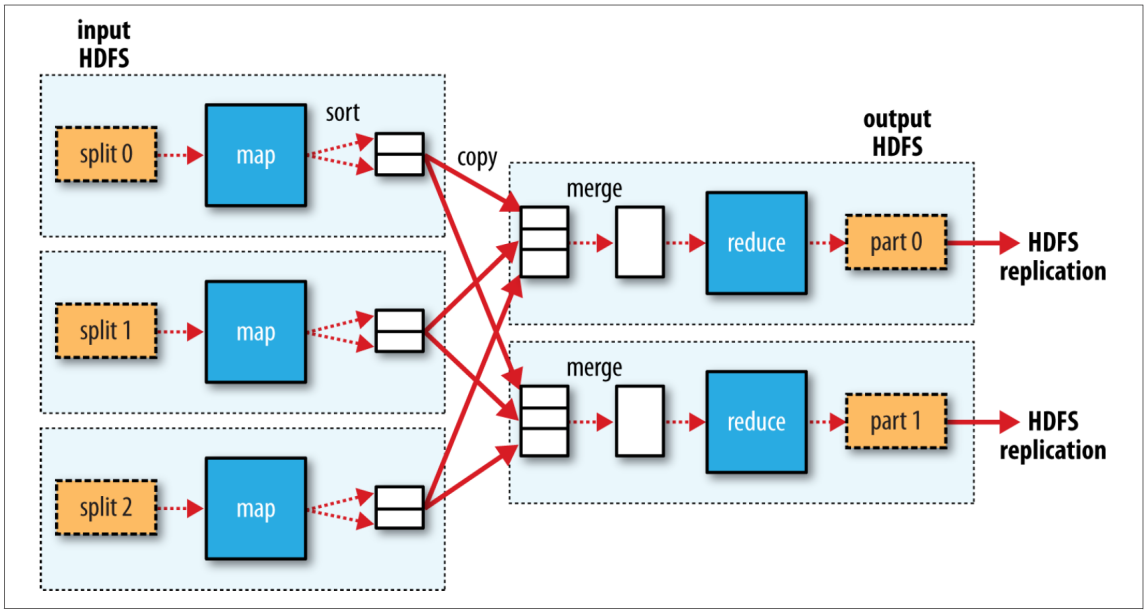


Figure 1.1: MapReduce pipeline [62]

Hadoop is an open source framework for distributed processing of large datasets using a cluster of computers. The framework implements the MapReduce programming model for large scale data processing across clusters of commodity machines, and uses the Hadoop Distributed File System (HDFS) [51], which is a high-throughput and reliable distributed file system for easy storage and retrieval of large datasets across multiple machines. Hadoop provides high-availability and fault tolerance in case of node failures. The initial version of Hadoop is tightly coupled to the MapReduce programming model; today's big data processing applications require new and more complex data processing engines that can handle modern workloads such as interactive SQL, real-time streaming, machine learning, and graph processing.

Although MapReduce is very powerful for distributed batch processing of large datasets, it does not perform well for two classes of applications: iterative algorithms that perform a set of operations multiple times on the same subset of a dataset such as machine learning applications, and interactive data mining where users runs multiple queries over the same subset of data. This is because MapReduce shuffles intermediate data to an external storage e.g. disk, therefore, for applications that reuse intermediate data multiple times, disk or network I/O would become a bottleneck. In these applications, keeping intermediate data in-memory

can substantially improve performance. To address this, researchers proposed Resilient Distributed Datasets (RDDs) [68] that allows applications to keep intermediate data in memory, in a parallel and fault-tolerant way, and reuse the data multiple times, therefore, providing a substantial performance improvement. RDDs are the basic data abstraction in Apache Spark.

Spark [69] is a general engine for large-scale distributed data processing that supports both batch and streaming processing. Using in-memory computation, Spark increases its performance over other frameworks, such as Hadoop, in two types of computation: 1) iterative algorithms that reuse intermediate results across multiple computations and 2) interactive data mining where users run multiple queries over the same subset of data. Spark provides high-level APIs in different programming languages including Java, Scala, Python and R, making it easier to write distributed image processing applications using several image processing libraries already implemented in these languages. It also supports higher-level tools including Spark SQL for SQL and structured data processing, MLlib for machine learning, GraphX for graph processing, and Spark Streaming.

1.2 Distributed Image Processing

The advancement of image acquisition tools such as modern cameras, sensors, satellites and drones has made capturing of large scale image data possible. This provides great opportunity for both science and industry to apply sophisticated image processing, statistical, and machine learning algorithms on large scale images to extract useful information, which could be used for a vast number of applications in areas such as Agriculture, Geographic Information System, Medicine and Security. As the volume of data becomes large, the main concern is how to store and process the generated data in a cost and time efficient manner. Modern image processing algorithms such as image registration [73], image segmentation and object recognition [4] are complex and resource intensive. Fast and efficient execution of these algorithms on a large dataset requires substantially more computation and storage power than that of a traditional desktop computer.

1.3 Thesis Motivation

Distributed processing of large datasets across multiple nodes on a cluster has been gaining popularity in recent years. For such large clusters, multiple applications with different resource requirements could run along side each other concurrently. Therefore, it is very important to ensure that multiple applications are able to share cluster resources efficiently in order to achieve maximum cluster resource utilization and minimize jobs makespan on shared clusters.

This thesis proposes using Apache Spark to speedup processing of large image data in a time and cost effective manner. Although some stages within the workload applications used in this thesis do not necessarily make use of the main advantage of Spark - keeping intermediate data in memory, these sets of applications

still benefit from using other features such as distributed processing and optimization of the implementation of various machine learning algorithms.

Among the many distributed data processing frameworks available, Spark was chosen for implementing and executing the workload applications used in this thesis: Spark is a stable, popular and widely used engine for large-scale distributed data processing. Furthermore, Spark has the advantage of providing supports for many modern programming languages including Python - the language in which the sequential versions of the applications are originally written. This substantially reduces the software development process of converting the sequential Python code to Spark, and provides the ability to use the same processing pipeline and easily integrates with other libraries available in Python. Spark provides a Standalone cluster manager as part of its distribution, which implements a simple FIFO scheduling for multiple concurrently running applications. In addition, Spark applications can be submitted to Apache Mesos [25] and Hadoop YARN [56] clusters. Mesos was started in 2009 with the aim of improving resource utilization by dynamically sharing resources among multiple frameworks (such as Spark applications). YARN was introduced in next generation version of Hadoop [62], and provides many improvements over the initial version. The basic idea behind YARN is to separate the programming model from job scheduling and cluster resource management. This makes it easy to port and run several other programming models such as Spark on top of YARN.

In contrast, the Hadoop framework is written for Java. There are no Python APIs provided by Hadoop out of the box at the time of writing this document. Applications written in Python need to be translated using third party libraries into a Java jar file. In addition, other third party Python frameworks exist for working with Hadoop. It is difficult, however, to provide a fair experimental performance analysis of the workload applications in Hadoop. These experiments and analysis are beyond the scope of this thesis.

Cluster administrators are responsible for the scheduling and allocation of cluster resources (i.e. CPU cores and memory) across multiple applications. This leads to performance issues because they must decide which cluster manager is the best fit for the applications. In addition, they must understand the benefits and overhead of enabling dynamic resource allocation on top of the different supported cluster managers, and how the various scheduling and resource allocation parameters of Spark affect the performance of multiple concurrent applications in terms of makespans and overall cluster resource utilization. Therefore, there are two motivations for this work:

- The first motivation is to parallelize a set of heterogeneous image processing applications to see if they are properly amenable to the Spark platform and in particular to being executed concurrently, and also investigate the amount of resources that are required for the distributed version of the applications running on Spark to compare with the sequential version running on a server equipped with large amount of resources. Although some of the stages in the image processing applications do not make use of the various characteristics and benefits of Spark, such as keeping intermediate data in memory, the experiments in this thesis will attempt to identify if distributed processing of these applications can provide substantial performance improvement over the traditional sequential processing.

- The second motivation is to perform an in-depth analysis of the performance implication and potential overhead of enabling dynamic resource allocation on top of the three different Spark supported cluster managers including Standalone, Mesos and YARN, and how different parameters affect the performance of multiple heterogeneous image processing applications in terms of makespan and the overall cluster resource utilization.

1.4 Thesis Statement

This thesis intend to determine whether the Spark cluster managers can be configured to reduce the makespan for image processing applications, and improve cluster resource utilization. As a case study, three applications, Image Registration, Flower Counter, Image Clustering, originally written in Python will comprise the cluster workload. A set of performance experiments with various scenarios will evaluate quantitatively the performance impact of Spark configuration parameters that control resource scheduling and allocation, number of parallel tasks per job, speculative execution of tasks and resource size per Spark executor, in terms of selected performance measures, including jobs makespan, CPU and memory utilization and jobs waiting and execution times.

1.5 Thesis Organization

The rest of the thesis document is organized as follows. Chapter 2 describes the background and reviews related work. Chapter 3 gives a detailed description of the hardware and software configurations of the testbeds used for evaluation, as well as the design of various experiments used in the thesis. Chapter 4 presents the results of the performance of image processing applications on different Spark cluster managers, under various job scheduling and resource allocation parameters settings. In this chapter, the results of running both the sequential and distributed version of the workload applications on the testbeds are compared and analysed. Finally, Chapter 5 presents the conclusions and contributions of the thesis along with future work.

Chapter 2

Background and Related Work

This chapter provides a detailed description of Apache Spark, including the components of a Spark application, data access from HDFS, different cluster managers that support Spark and the various policies they use for node management and resource allocation. The first part of the chapter defines and explains some of the important terms used in this thesis, as well as provides an overview of the Spark architecture, its resource structure, how applications are submitted and run, the different resource allocation modes available in Spark and how to enable them, and the various parameters that control Spark behaviour. The second part of the chapter discusses the three Spark cluster managers including Spark Standalone, Apache Mesos, and Hadoop YARN and their architecture and resource allocation policies. Apache Mesos and Hadoop YARN cluster managers are not exclusively used with Spark, but are available to allow many BigData Analytics applications to share cluster resources efficiently. Finally, the last part of the chapter presents a literature survey and describes related work.

2.1 Background

2.1.1 Definitions

This section provides definitions of the various terms used in this thesis. Most of the material provided has been adapted from the Spark online documentation page.[1]

Resilient Distributed Datasets (RDDs):

Resilient Distributed Datasets [68] are a fault-tolerant, distributed memory abstraction for in-memory processing of large datasets on a cluster. The RDD is the primary data abstraction in Spark. It represents an immutable, partitioned collection of elements that can be operated on in parallel. An RDD stores information about how it was computed (its lineage), and uses this information to reconstruct partitions from the datasets in case of failure.

[1] `https://spark.apache.org/docs/2.1.0/` (2 Mar, 2018)

Application:

A Spark *Application* is a self-contained unit of computation (i.e. program) that runs user-supplied code written using the Spark API to process data. It consists of a single driver process and a set of executor processes distributed across hosts in a cluster. A Spark application can be a single batch job, an interactive session with multiple jobs, or a long-lived server process running and continually satisfying user's requests.

Driver program:

A Spark *Driver* program is a process that creates a *SparkContext* and runs the *main* method of a Spark *Application*. The *Driver* is the process that runs user code which creates RDDs, and performs transformation and actions. It is also responsible for converting the Spark *Application* into tasks and scheduling the tasks to run on the executors allocated to the application by the cluster manager. The task scheduler resides in the *Driver* and distributes tasks among worker nodes. Therefore, the two important roles of a *Driver* program are converting user programs into tasks and scheduling tasks on executors.

SparkContext:

A *SparkContext* is the main entry point for a Spark *Application*. It creates the connection to a Spark cluster and allows the Spark *Driver* program to access the cluster through a cluster manager. It can also be used to create RDDs, accumulators and broadcast variables on the cluster. When a Spark *Application* starts, the first thing it does is create a *SparkContext* object that contains information about the application, and tells Spark how to access the cluster. Only one *SparkContext* may be active per Spark *Application's* JVM at any given time.

SparkConf:

A *SparkConf* is a Spark object that stores a list of configurations for a Spark *Application* defined by the application programmer. The *Driver* program creates *SparkContext* and passes these configurations to it as key-value pairs that define application properties, such as application name and required resources. When a *SparkConf* object is created, it loads values from any spark-related Java system properties set in the application. Furthermore, parameters set directly on the *SparkConf* object take priority over system properties.

Executors:

Executors are processes launched on worker nodes and are responsible for executing the individual tasks in a given Spark *Job* and returning results to the *Driver* program. In addition, they provide in-memory storage for RDDs that are cached by user programs, through a service called *BlockManager* that lives within each executor. RDDs are cached directly inside of *Executors*, therefore, tasks can run alongside the cached data. Each Spark *Application* consists of a set of *Executor* processes that stay alive for the duration of the

application and run tasks in multiple threads. This provides the benefits of isolating applications from each other, since each Spark *Application* gets an independent set of *Executor* JVMs that only run tasks and store data for that application.

Job:

A *Job* is a set of computations, consisting of multiple tasks grouped into one or more stages and submitted to Spark *DAGScheduler* for computing the result of an action in parallel on different executors. A Spark *Job* is triggered in response to Spark actions such as *Map, Collect* and *Count* and spawns multiple tasks. When a Spark job is triggered, an execution plan is created according to the lineage graph. The executing *Job* is then split into stages, where stages contain multiple transformations and actions. Generally, a *Job* computes the partition of a single RDD by applying different Spark actions (i.e. computations). However, to compute the partitions of an RDD, a Spark *Job* can also use other RDDs that are part of the target RDD's lineage graph (i.e. all its parent RDDs).

Stage:

Spark jobs are divided into *Stages*. A *Stage* is a physical unit of execution which consist of a set of smaller parallel tasks (i.e. one task per RDD partition) of a particular *Job* grouped together. *Stages* are divided based on computational boundaries, and all tasks within a particular stage depend on each other and can only work on the partitions of a single Spark RDD. However, a *Stage* can be associated with many other dependent parent stages. Therefore, submitting a *Stage* can trigger execution of a series of dependent parent stages.

Tasks:

Tasks are the smallest individual unit of execution that can run on *Executors*. Each *Stage* in a Spark *Application* consists of a set of tasks that are scheduled and send from a *Driver* program to an *Executor* by serializing a function object. The *Executor* de-serializes the command, and executes it on different partitions of a Spark RDD. Tasks are spawned one by one for each *Stage* and partition. In addition, a *Task* can only belong to one *Stage* and operate on a single RDD partition. Furthermore, all tasks in a *Stage* must be completed before the stages that follow can start.

Cluster manager:

Spark relies on cluster managers to launch executors [32]. Cluster managers are pluggable, external services responsible for scheduling and allocating cluster resources such as CPU and memory to applications. Each Spark application runs an independent set of processes on the cluster. These processes are coordinated by the application's *SparkContext* which connects to the three different cluster managers that support Spark, including Spark Standalone, Apache Mesos and Hadoop YARN.

PySpark:

Spark was originally written in Scala, a language that compiles to bytecode for running on Java Virtual Machine (JVM) [45]. However, the open source community has developed a toolkit called *PySpark*[2] that uses *Py4J*[3] library to interface with Spark's RDDs (which are JVM objects). *PySpark* is a Spark Python API that exposes the Spark programming model to Python. This API allows programmers to perform various transformations and actions on Spark's RDD using Python.

Image processing applications such as the benchmark applications used in this thesis are easily implemented in Python[4] using various open source libraries such as the open source computer vision library - *OpenCV*.[5] Moreover, the PySpark API easily allows programmers to parallelize applications that are already written in Python to take advantage of the extremely high-performance data processing enabled by Spark's Scala architecture. This is especially important to programmers that are already familiar with Python because it allows them to write programs in the language they are most familiar with, perform distributed transformations on large datasets and retrieve results in a Python-friendly notation. In addition, image processing applications implemented using PySpark can take advantage of other Python based scientific computing libraries such as *NumPy* for fast and efficient processing of large binary datasets.

Hadoop Distributed File System (HDFS):

The Hadoop Distributed File System [51] is a high throughput and fault-tolerant distributed file system that is designed to be deployed on inexpensive commodity machines. It is one of the modules included as part of the Hadoop distribution and is suitable for storing large datasets. The HDFS has a master/slave architecture, with a single master called the NameNode that manages the file system namespace and regulates access to files by clients, and many DataNodes that manage local storage attached to the nodes.

NameNode:

The *NameNode* is the central component of the HDFS file system [62]. The *NameNode* does not store the files data itself; rather, it stores the directory tree of all files in the file system, and keeps track of the locations across the cluster where files are stored. Client applications talk to the *NameNode* whenever they wish to locate a file on the file system such as during operations like add/copy/move/delete. When applications need access to a file stored on HDFS, the *NameNode* returns a list of *DataNodes* where the data is stored.

[2] `http://spark.apache.org/docs/2.1.0/api/python/` (2 Mar, 2018)
[3] `https://www.py4j.org/` (2 Mar, 2018)
[4] `https://docs.python.org/2.7/` (9 Apr, 2018)
[5] `https://opencv.org/` (9 Apr, 2018)

DataNode:

DataNodes store actual data in the file system [62]. A functional file system has multiple *DataNode* instances on a cluster, which communicate with each other and replicate data. Applications can also directly communicate with *DataNodes*, once the *NameNode* has provided data location.

2.1.2 Overview of Apache Spark

Figure 2.1 shows the various core components of Apache Spark. A summary of each of the core Spark components is provided below. An extensive description can be found in Karau *et al.* [32]. The workload applications used in this thesis utilize many of the Spark Core API *transformations* and *actions*. Other Spark components including *Spark SQL, Spark Streaming, MLlib,* and *GraphX* are not used in this thesis but a brief description is provided for completeness.

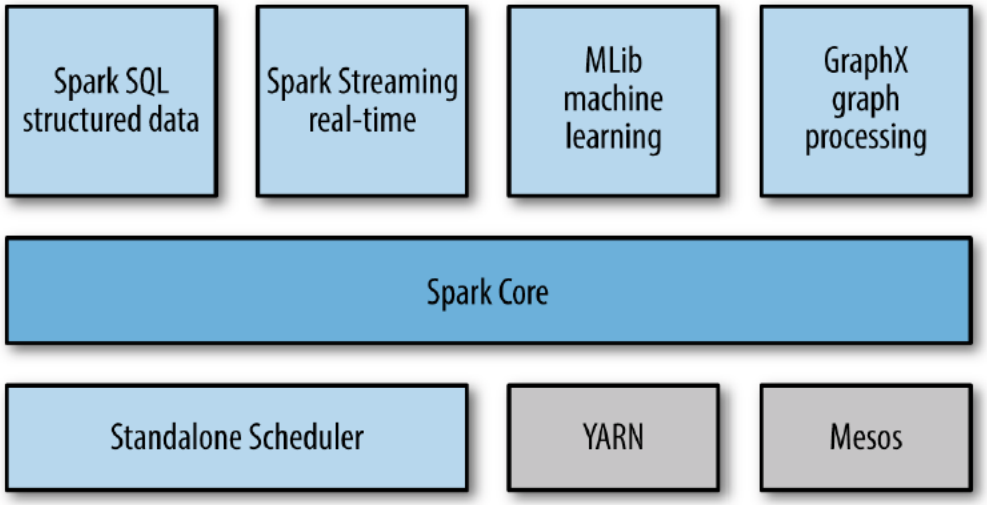


Figure 2.1: Spark architecture [32]

Spark Core API:

The Spark Core component is the foundation for Spark's data processing. It handles basic functionalities such as scheduling and monitoring of jobs, efficient memory management, fault recovery and interacting with various data sources. Spark core also defines APIs for Resilient Distributed Datasets (RDDs), and the different types of transformations such as *map, filter, reduceByKey* which creates a new RDD from existing data, and actions such as *reduce, collect, count* which perform a computation on an RDD and return a value.

Spark SQL:

The Spark SQL is a library built on top of Spark to provide facilities to perform SQL like queries such as extract, transform and load (ETL) on Spark RDDs and other data sources including Hive[6] tables, Parquet,[7] and JavaScript Object Notation (JSON).[8] The library allows developers and users to run ad hoc queries using Structured Query Language (SQL) and Hive Query Language (HQL), therefore, easing the process of extracting and merging various datasets and making them available for other applications such as machine learning.

Spark Streaming:

Spark Streaming is a high throughput, fault tolerant and scalable API that enables processing of live streams of data from various data sources such as Kafka,[9] Flume[10] and HDFS. The API uses the Discretized Stream objects, which represents a continuous stream of data divided into small batches. Discretized Stream is built on Spark's Core RDD's API, therefore, RDD transformations and actions can be applied to the mini batches of the input data for real-time processing.

MLlib:

Spark's MLlib is a machine learning library used for different types of machine learning applications such as classification, regression, clustering, and collaborative filtering. It implements many commonly used algorithms including Naive Bayes, Logistic Regression, Support Vector Machine, Linear Regression and K-means, as well as lower-level machine learning primitives including a generic gradient descent optimization algorithm. It also provides support for data import and model evaluation.

2.1.3 HDFS and Spark RDD Partitions

The Hadoop Distributed File System (HDFS) use the concept of blocks to store files [51]. When a user stores a large file in HDFS, the *NameNode* divides this file into various chunks and stores these chunks in blocks on different *DataNodes*. When a user stores a file in HDFS that is smaller than the block size, however, the file will only occupy the amount of the underlying physical storage it requires, not the entire block.

HDFS is designed to store large datasets at the terabytes scale; therefore, it is not optimal for storing large number of small files [62], that are considerably smaller than the specified HDFS block size. There are several reasons for this.

Firstly, a large number of small files can populate HDFS namespace quickly, and because there is namespace limitation, this could result in underutilizing disk space. Secondly, every HDFS block has metadata

[6] `https://hive.apache.org/` (9 Apr, 2018)
[7] `https://parquet.apache.org/` (9 Apr, 2018)
[8] `https://www.json.org/` (9 Apr, 2018)
[9] `https://kafka.apache.org/` (9 Apr, 2018)
[10] `https://flume.apache.org/` (9 Apr, 2018)

that the *NameNode* records and stores in memory. Therefore, storing a large number of small files increases the total number of blocks and the metadata the *NameNode* must handle in RAM.

Furthermore, for MapReduce applications, Hadoop tries to start one mapper per HDFS block on the machines where the data is stored to minimize network transfer of data from the storage node to the processing node. Large files are more likely to be stored on the same cluster node as opposed to small files which can be spread-out across the cluster. Having one mapper per small block could result in resource wastage and increase latency, because there is an overhead to start a mapper, and these mappers would only process a small amount of data, and eventually be allocated to other tasks by the scheduler.

When reading data from a distributed file system such as HDFS, Spark tries to utilize the file system's distributed data partitions to determine the number of RDD partitions to create. During execution, Spark creates RDD partitions on the node containing the data block and allocates data processing (typically a single task per RDD partition) with minimum data transfer across the network. Spark provides various transformations such as *repartition, coalesce* and *repartitionAndSortWithinPartition* that could be used to re-assign the number of RDD partitions at runtime.

2.1.4 The Concept of Resource in Spark

As mentioned in Section 2.1.1, executors are the resource unit in Spark. They contain a number of CPU cores and memory, which are controlled by *spark.executor.cores* and *spark.executor.memory* respectively. By default, each executor process is made up of 1 GB memory in all cluster managers, 1 CPU core in YARN mode, and all the available cores on the worker in Standalone and Mesos modes. Each task occupies *spark.task.cpus* number of CPU cores. Therefore, by tuning these parameters, the number of tasks that could simultaneously run on an executor may be adjusted, thereby improving the overall cluster utilization. Table 2.1 shows some of the important scheduling parameters that could be configured in Spark to improve resource utilization and job response times.

2.1.5 Spark Runtime Architecture

Spark uses a uniform interface to submit an application to every supported cluster manager. The arguments passed to this submission script determine from which cluster manager (Standalone, Mesos, YARN) to request resources. The script also loads the default Spark configuration values from the configuration file and passes them on to the application. Generally, configuration values explicitly defined by the application's *SparkConf* take the highest precedence, then arguments passed to the submission script, then values in the default configuration file.

A code snippet for WordCount application written in PySpark is shown in Listing 2.1. This application reads in a text file from HDFS and creates a Spark RDD of the file. The application then counts how often each word occurs in the RDD. The output is a series of text files written to HDFS. Each line in the output files contains a word and the count of how often it occurred.

Table 2.1: Spark Schedulling Configuration Parameters

Configuration parameter	Default value	Description
spark.cores.max	(not set)	Maximum number of CPU cores to request across the cluster
spark.deploy.defaultCores	(infinite)	Used if applications do not set *spark.cores.max*
spark.executor.cores	1 core in YARN mode, all cores on the worker in Standalone and Mesos mode.	The number of cores to use for each executor process.
spark.executor.memory	1 GB	Amount of memory to use per executor process
spark.executor.instances	2	The number of executors for static allocation in YARN mode, or the initial set of executors to allocate when using dynamic resource allocation
spark.task.cpus	1	Number of cores for each task
spark.files.maxPartitionBytes	128 MB	Maximum partition size
spark.speculation	false	If set to *true*, slow running tasks in a stage may be re-launched
spark.speculation.interval	100ms	How often Spark will check for tasks to speculate
spark.speculation.multiplier	1.5	How many times slower a task is than the median to be considered for speculation
spark.speculation.quantile	0.75	Fraction of tasks which must be complete before speculation is enabled for a particular stage

Listing 2.1: WordCount in PySpark [3]

```
"Import the necessary libraries"
from pyspark import SparkConf, SparkContext

"Read a text file from HDFS and create an RDD from the file"
sc = SparkContext(appName="WordCount")
text_file = sc.textFile("hdfs://...")

"Count the occurrence of each word in the RDD"
counts = text_file.flatMap(lambda line: line.split(" ")) \
            .map(lambda word: (word, 1)) \
            .reduceByKey(lambda a, b: a + b)

"Save the result back to HDFS"
counts.saveAsTextFile("hdfs://...")
```

On application submission, Spark connects to the specified cluster manager to acquire executors on nodes in the cluster. The number of executors to request depends on the number of tasks in the application and the configuration values set to control resource size per executor. For example, if *spark.task.cpus* configuration parameter is set to 1 and each executor is set to have 1 CPU core using *spark.executor.cores* configuration parameter, then 1 executor would be requested from the cluster manager per task, as long as the total number of allocated cores does not exceed *spark.cores.max* or *spark.deploy.defaultCores* for applications that

do not set *spark.cores.max* parameter. The cluster manager allocates executors to the application based on availability and configuration values. Spark then sends the application code to the executors, this allows *SparkContext* to run tasks on these executors.

In this architecture, each Spark application gets its own set of executor processes, and holds on to the executors for the duration of the whole application. The application's *SparkContext* schedules tasks on the cluster and runs these tasks in multiple threads. Both the driver program and the executors communicate with each other throughout the lifetime of the application. This lets Spark isolate applications from each other both on the scheduling side, since each driver schedules its own tasks, as well as the executor side because tasks from different applications run in different JVMs. However, data can not be shared between different applications without first writing it to an external storage.

2.2 Spark Cluster Managers

Distributed computing has been the centre of modern data processing in many big organizations. In this type of architecture, a large number of commodity computers are grouped together to form a cluster of computers. These clusters are often shared by many applications with varying resource requirements. Therefore, there is the need for efficient resource sharing in order to achieve high resource utilization. Cluster managers are responsible for the scheduling and allocation of cluster resources (CPU and memory) across multiple applications.

Table 2.2: Characteristics of the Cluster Managers Tested

Cluster manager	**Spark Standalone**	**Apache Mesos**	**Hadoop YARN**
Node management	A single master manages multiple worker nodes	A single master manages multiple slave nodes	A *ResourceManager* daemon runs on the master node, and a *NodeManager* daemon runs on each slave node
Resource allocation	Schedule multiple applications using a *FIFO* scheduler	Schedule multiple applications using a scheduler that implements the Dominant Resource Fairness (DRF) algorithm	Implements three algorithms for scheduling multiple applications including *FIFO*, *Fair* and *Capacity*
High availability	Resilient to worker failure and provide high availability for the master node using ZooKeeper	One active and multiple standby masters may be configured using ZooKeeper	Allow multiple standby *ResourceManagers* for automatic recovery via Zookeeper in case of failure
Resource isolation	Does not take care of resource isolation	Uses existing OS isolation mechanisms such Linux Containers to isolate executors of different frameworks	Enforces memory limits for all containers. Uses CGroups to isolate and limit container's CPU usage

This section gives an overview of the different cluster managers available in Spark, and looks in more detail at their nodes management, resource allocation, high availability, and resource isolation. Spark does not need to be aware of the underlying cluster manager it runs on, as long as applications are able to acquire executor

processes, and these executors can communicate with each other. This enables Spark runs on different cluster managers. Figure 2.2 shows how applications run on Spark. A summary of the various characteristics of the three cluster managers evaluated in this thesis is provided in Table 2.2.

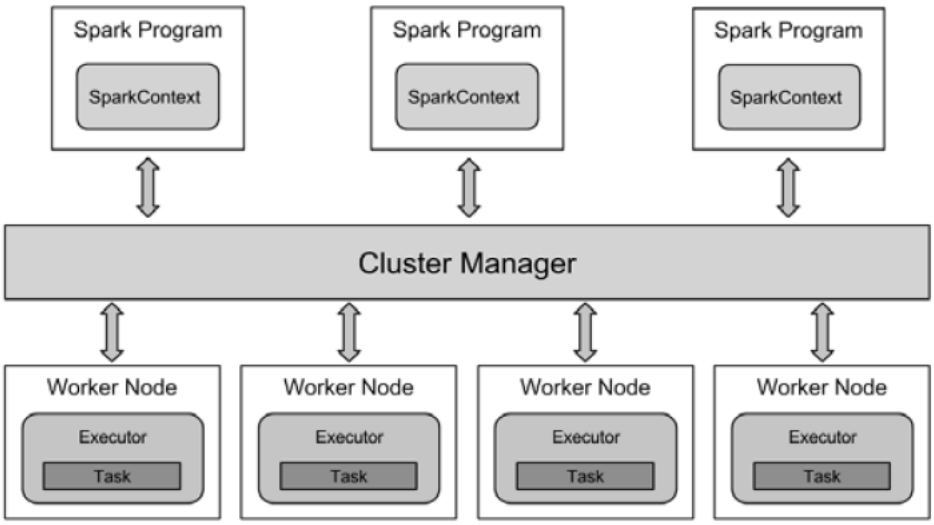


Figure 2.2: Running applications on Spark [31]

2.2.1 Spark Standalone

The Spark Standalone cluster manager is a simple cluster manager available as part of the default Spark distribution. It has high availability for the master, is resilient to worker failures, has capabilities for managing resources per application, and can run alongside an existing Hadoop deployment and access HDFS data.

Node Management

Standalone uses a single master to manage and coordinate multiple worker nodes. Each worker node is configured to provide the cluster with a certain amount of CPU and memory resources. Applications submitted can specify the amount of the resources (CPU cores and memory) they need per executor from the cluster manager.

Resource Allocation

Applications submitted to the Standalone cluster manager are scheduled in FIFO order, and each application will try to use all available nodes by default. Subsequent applications submitted will be queued and must wait for the first application to run to completion before they can start running jobs, even if the first application is not using all the resources allocated to it, unless the appropriate configuration parameters are set. The Standalone cluster manager currently only supports FIFO scheduling.

To allow multiple concurrent users, application developers must set *spark.cores.max* configuration parameter within their applications to limit the number of cores the applications request across the cluster. Moreover, the cluster administrator may set *spark.deploy.defaultCores* parameter to set the maximum number of cores each application submitted to the cluster may request. Setting these configurations will prevent applications from monopolizing the entire cluster when using static resource allocation on the Standalone cluster manager.

High Availability

The standalone cluster manager is resilient to worker failure, since applications can simply restart the tasks that were lost on other available workers. Spark can be configured to launch multiple Masters in the same cluster, connected to a ZooKeeper [28] instance. In the event of master failure, ZooKeeper (a distributed coordination system) selects a new master from the list of standby masters in a timely manner, a process called leader election. The new Master would recover the old Master's state, and then resume scheduling. Master failure only affects scheduling of new applications. The Spark cluster can also use Local File System to recover applications and workers upon a restart of the master process.

Resource Isolation

The Standalone cluster manager only handles resource allocation, including keeping track of available and allocated resources. It does not take care of resource isolation.

2.2.2 Apache Mesos

Apache Mesos [25] is a distributed, highly available, and fault-tolerant cluster manager that is designed to act as the kernel for dynamically sharing cluster resources among multiple frameworks. Mesos abstracts cluster resources from multiple nodes in a cluster, including CPU, memory, and disk and allows them to be viewed and used between multiple frameworks as if they were a single, very large server [29]. Mesos uses containers to isolate processes from different applications, therefore allowing applications to run alongside each other on the same computer without interfering. Figure 2.3 shows how Mesos abstracts cluster resources and shares them between different applications. At the lower lever are the physical machines, consisting of both computing and storage resources. The OS kernel provides access to underlying physical or virtual resources. The Mesos kernel runs on top of the OS kernel on every machine in the cluster and abstracts the physical resources of these machines using the same principles as the OS kernel, only at a different level of abstraction.

Node Management

Mesos uses a master/slave architecture to manage cluster nodes. In this architecture, a single master process manages multiple slave processes running on each cluster node. The master process provides resource offers to frameworks and launches tasks on the slave nodes. It also handles communication between tasks

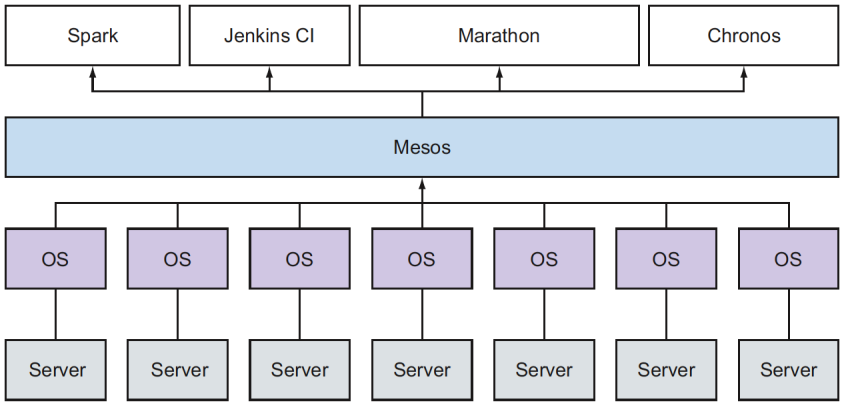


Figure 2.3: Running applications on Mesos [29]

and frameworks. The slave nodes manage physical resources such as CPU, memory and disk, and execute framework tasks. Slave nodes are Mesos's equivalent of Spark's worker nodes. Frameworks are applications that run on Mesos (e.g. Hadoop and Spark) and consist of two parts, namely the scheduler and the executor. The scheduler negotiates with the master and handles resource offers and task monitoring, while executors accept and process tasks. Since resource scheduling is the responsibility of both the Mesos master's allocation module and the framework's scheduler, this is referred to as two-level scheduling. Mesos slaves periodically report their available resources to the master's allocation module. The master makes decisions as to which framework to offer the available resource and advertises this resource offer to the selected framework. The framework can decide to reject the offer if it does not satisfy its resource requirements. On accepting the offer, the framework's scheduler sends tasks and resources to use for each task to the master, which finally sends the tasks to slaves for execution.

Resource Allocation

Mesos provides organizations with the ability to define their own resource allocation policies using a pluggable allocation module. By default, Mesos uses a fair-sharing resource allocation algorithm called Dominant Resource Fairness (DRF) [17] to allocate resources to frameworks. The DRF algorithm aims to share multi-dimensional resources (e.g. CPU and memory) across multiple users. A framework's dominant resource is the most demanded resource type for that framework such as CPU, memory or disk. DRF computes the framework's total share of this resource (dominant share), and tries to maximize the minimum dominant share in the system. For example, if user A runs CPU-intensive tasks and user B runs memory-intensive tasks, DRF attempts to equalize user A's share of CPUs with user B's share of memory. This implies that user A would get more CPU and less memory, while user B would get less CPU and more memory. Whenever there are available resources and tasks to run, the scheduler selects the user with the smallest dominant share and allocates resources to it first.

High Availability

All Mesos slaves and running frameworks depend on Mesos master to manage the slave nodes and offer resources to active frameworks. In order to prevent cluster failures, Mesos provides the ability to configure multiple masters, with one active master, and the remainder operate on standby. The active master stores only the list of active slaves, active frameworks, and running tasks. This information can be used to compute the number of resources allocated to each framework and the allocation policy. In the event of a master failure, Mesos uses ZooKeeper to select a new master from the list of standby masters.

Resource Isolation

Mesos uses existing OS isolation mechanisms such Linux Containers to limit resources (CPU, memory, network bandwidth) of frameworks and isolate executors of different frameworks running on the same Mesos slave.

2.2.3 Hadoop YARN

The Hadoop YARN [56] cluster manager is part of the second-generation Hadoop version 2 distribution. YARN allows multiple data processing engines with diverse programming models to run alongside Hadoop MapReduce on the same cluster and have access to centralized datasets stored on HDFS [51]. In YARN, resource management and job scheduling/monitoring are handled by separate daemons consisting of global *ResourceManager* and per-application *ApplicationMaster* [44]. Figure 2.4 shows the high-level YARN architecture.

The *ResourceManager* daemon runs on the master node and arbitrates resources among all applications. It consists of a Scheduler which is responsible for allocating resources to applications, and *ApplicationManager* which is responsible for accepting job submissions and negotiating the first container to run *ApplicationMaster*. The per-application *ApplicationMaster* is responsible for negotiating container resources from the *ResourceManager's* Scheduler, tracking the status of allocated containers and monitoring tasks execution progress.

Node Management

A *NodeManager* runs on each slave node; it is responsible for managing containers running on the node. It also monitors container's resource usage such as CPU, memory, disk and network, and it reports their status to the *ResourceManager*.

Resource Allocation

YARN's *ResourceManager* contains a pluggable global scheduler which is responsible for partitioning and allocating cluster resources to various applications. By default, YARN implements different schedulers in-

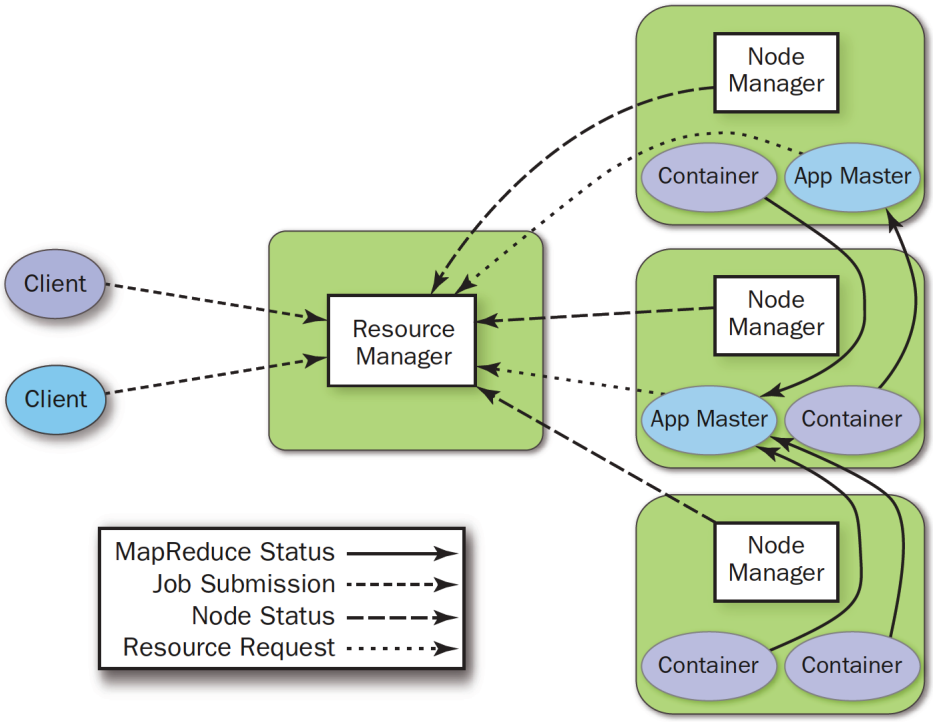


Figure 2.4: Running applications on YARN [44]

cluding the previously mentioned *FIFO Scheduler*, *Fair Scheduler* which aims to share cluster resources fairly among all running jobs, and *Capacity Scheduler* which allows sharing of a Hadoop cluster among organizations by providing each organization with a guaranteed share of overall cluster capacity. Organizations can also implement their own resource allocation policies, however, as per requirements and resource availability.

High Availability

High availability is achieved in the YARN cluster manager through an Active/Standby architecture. At any point in time, there is only one active *ResourceManager*, and one or more in standby mode waiting to take over in case the active *ResourceManager* fails. In the event of failure, automatic recovery via Zookeeper can be enabled. In addition, cluster administrators can trigger an action to transition one of the standby *ResourceManagers* into active mode through the Command Line Interface (CLI). On the other hand, when the *ApplicationMaster* fails, the *ResourceManager* detects this failure and starts a new instance on another container. In the event of a container failure, the *NodeManager* detects this failure event and launches a new container replacement. It also restarts all tasks execution on the new container.

Resource Isolation

YARN automatically enforces memory limits for all containers that it controls. Containers must have a maximum memory size defined when created. If the sum of all memory usage for processes associated with a

single YARN container exceeds this maximum, YARN will kill the container. YARN uses CGroups to isolate and limit container's CPU usage, since this must be done at the kernel level. CGroups enables users to set limits on the amount of CPU resources granted to individual processes such as YARN containers.

2.3 Resource Allocation Modes

2.3.1 Static Resource Allocation

By default, Spark uses *Static Resource Allocation* to share resources on all the three supported cluster managers. With this approach, applications are assigned resources statically, and hold on to them for the duration of their lifetime. The Spark execution model allows applications to reuse resources by having a long running thread which resides in the executor process. This provides many performance optimizations relating to speed which include: efficient data sharing in memory, fine-grained partitioning and low latency scheduling because there is no need to start a new executor process for each newly submitted task; instead a long running process accepts and runs many small tasks. However, this speed-up gain comes with a cost, since idle applications may hold on to resources for a very long time, even if they are not being used.

To limit the amount of resources applications use per executor process, the application programmer needs to set the appropriate configuration parameters (*spark.executor.cores* and *spark.executor.memory*) within their application to specify the number of cores and memory to request per executor from the cluster manager. The default value for *spark.executor.cores* is set to 1 in YARN mode and all available cores on the worker machine in Standalone and Mesos coarse-grained modes. The value of *spark.executor.memory* is set to 1 GB in all the three cluster managers by default. Using the default values for these parameters means that only one executor (with 1 GB memory and all available CPU cores on the worker machine) per application will run on each worker in Standalone and Mesos modes. On the other hand, applications submitted to a YARN cluster manager would request two executor instances for running tasks by default, each having 1 CPU core and 1 GB memory regardless of the available resources in the cluster. Setting the appropriate values for the configuration parameters that control resource size per executor would allow applications to run multiple executors on the same worker node.

Users can also use *spark.cores.max* to set the maximum number of CPU cores to request for a single application from across the cluster in Standalone and Mesos mode. On the YARN cluster manager, the *spark.executor.instances* controls how many executors to allocate on the cluster. Cluster administrators may set *spark.deploy.defaultCores* parameter to control the total number of cores to give to applications that do not set the maximum CPU resources they need. Statically assigning these values might result in unnecessary increase in latency for applications, and inefficient utilization of cluster resources. Overall, the resource scheduling problems introduced by using static partitioning include the following:

1. Underutilization of cluster resources: For example, an interactive application like *spark-shell* can request

resources it does not use. Meanwhile, other applications can not be executed because *spark-shell* is still running and holding on to the resources.

2. Starvation of other applications: If a single application occupies lots of resources without releasing them, other application will be queued.

3. Lack of elastic resource scaling ability: It is hard to estimate, with reasonable precision, the amount of resources an application will require for efficient execution. For example, in an iterative workload such as data exploration in which a user runs many short commands iteratively, static allocation of cluster resources would lead to inefficient utilization.

2.3.2 Dynamic Resource Allocation

To address the above issues, Spark provides a *Dynamic Resource Allocation* mode which dynamically adjusts the resources applications use based on the available workload. This means that applications may give resources back to the cluster if they are no longer used and request them later when they need to use them again. This is very useful in a multi-tenant shared cluster where multiple applications need to share cluster resources. This feature is available on all three Spark supported cluster managers considered in this thesis.

Resource Allocation Policy

Dynamic resource allocation requests and releases executors at run-time based on available workload. Spark provides various configuration parameters that control when and how to request more resources from the cluster manager when there are more pending tasks, and free these resources when they are no longer needed.

Request Policy: Applications using dynamic resource allocation would request more executors when there are pending tasks to run. Having pending tasks implies that the existing set of executors are insufficient to simultaneously run all tasks that have been submitted but not yet finished. Spark requests executors in rounds, the actual request is triggered when there have been pending tasks for *spark.dynamicAllocation.schedulerBacklogTimeout* seconds, and then triggered again every *spark.dynamicAllocation.sustainedSchedulerBacklogTimeout* seconds thereafter if the queue of pending tasks persists. The number of executors requested in each round increases exponentially from the previous round. For instance, an application will add 1 executor in the first round, and then 2, 4, 8 and so on executors in the subsequent rounds. The motivation behind this policy is a slow start in case only a few more executors are needed and exponential growth in case it turns out that many executors are actually needed.

Removal Policy: The removal policy is simple, Spark removes the executors that have been idle for more than *spark.dynamicAllocation.executorIdleTimeout* seconds, since executors should not be idle if there are still pending tasks to be scheduled.

Enabling Dynamic Resource Allocation

To enable dynamic resource allocation on Spark, two important configurations need to be set and multiple other optional settings that control the behaviour of this feature as shown in Table 2.3. First, applications need to set *spark.dynamicAllocation.enabled* and *spark.shuffle.service.enabled* to true. In addition, an *external shuffle service* must be set on each worker node in the cluster.

Shuffle in Spark is an all-to-all communication across the cluster. It involves a lot of data serialization, network IO, and disk IO because the shuffled data needs to be serialized and transferred over the network to other machines or written to a local file. When executing Spark operations that cause shuffling to occur such as *repartition*, *coalesce*, *groupByKey*, and *reduceByKey*, not all values for a single key necessarily reside on the same partition or node. As a result, Spark generates a set of map tasks to organize the data, and a set of reduce tasks to aggregate the shuffled data. The individual map tasks store their results in memory. Shuffle operations can consume a substantial amount of heap memory, however, because they utilize various in-memory data structures to organize records.

When data does not fit in memory, Spark sorts the data and spills it to disk. This introduces additional overhead of disk I/O and garbage collection. Furthermore, a shuffle generates a large number of intermediate files on disk which are preserved until the corresponding RDDs are no longer needed. This is done in order to prevent re-creation of shuffle files when the RDD lineage is re-computed.

Table 2.3: Spark Dynamic Resource Allocation Configuration Parameters

Configuration parameter	**Default value**	**Description**
spark.dynamicAllocation.enabled	false	Enable dynamic resource allocation
spark.dynamicAllocation.executorIdleTimeout	60s	Any executor idle for more than this duration will be removed
spark.dynamicAllocation.cachedExecutorIdleTimeout	infinity	Any executor idle for more than this duration and has cached data will be removed
spark.dynamicAllocation.initialExecutors	*spark.dynamicAllocation.minExecutors*	Initial number of executors
spark.dynamicAllocation.maxExecutors	infinity	Upper bound for the number of executors
spark.dynamicAllocation.minExecutors	0	Lower bound for the number of executors
spark.dynamicAllocation.schedulerBacklogTimeout	1s	If there have been pending tasks backlogged for more than this duration, new executors will be requested
spark.dynamicAllocation.sustainedSchedulerBacklogTimeout	*schedulerBacklogTimeout*	Same as *schedulerBacklogTimeout*, but used only for subsequent executor requests
spark.shuffle.service.enabled	false	Enables external shuffle service to preserve executors shuffle data

Normally, Spark executors exit either on successful completion or failure. In both cases, all the data written by the executor is no longer needed and can be safely discarded. On the other hand, using dynamic resource allocation allows executors to be removed when the application is still running. Removing the shuffle files written by this executor implies that a recompute of this data needs to be performed when the

application tries to access the files. This introduces unnecessary additional computation, thereby increasing the overall running time of the application. To solve this, *external shuffle service* allows executors to be safely removed without deleting their shuffle files. This is a long-running process that runs on each node of the cluster independently of Spark applications running and their executors. When *external shuffle service* is enabled, Spark executors will fetch shuffled files from it instead of from each other. This means any shuffle data written by an executor may continue to be served beyond the executor's lifetime. By default, dynamic resource allocation will not remove any executor that stores cache data either on disk or in memory. This can be configured by changing the default value of *spark.dynamicAllocation.cachedExecutorIdleTimeout*.

The concept of Spark's dynamic resource allocation works the same way regardless of the underlying cluster manager used. The main idea of this resource allocation mode is to dynamically adjust applications resource usage at runtime based on the current workload. Recall that Spark monitors the resource usage of each application. When an executor assigned to application has been idle for more than a certain duration, Spark simply issues a command to the cluster manager to remove the executor from the application. On the other hand, when Spark determines an application has backlog tasks, it issues a new request to the cluster manager for additional executors.

2.4 Related work

This section presents a summary of literature survey and describes the related work from two different perspectives: 1) *Job scheduling and resource allocation in distributed systems* and 2) *Image processing on distributed systems*.

2.4.1 Job Scheduling and Resource Allocation in Distributed Systems

Large-scale compute clusters are expensive to build, therefore, there is the need for efficient utilization of these clusters. This can be achieved by running a mix of workloads on the same machines such as a combination of CPU and memory intensive jobs, batch and real-time jobs etc. Although this reduces the amount of hardware required for a workload, efficient scheduling of these jobs is of major concern. In recent years, resource-aware scheduling has gained considerable attention. This section discusses on the various schedulers and cluster managers proposed in the literature, other than those discussed earlier in Section 2.2.

Cluster Managers

Quasar [15] designs resource efficient and QoS-aware cluster management which aims to improve overall cluster resource utilization in data centers by eliminating resource reservation and utilizing a performance-centric approach. It lets users specify their application requirements and performance constraints such as deadlines. This cluster manager tries to dynamically determine the right amount of resources to meet these constraints while maximizing the utilization of available resources. Taking user specified constraints such as application

deadlines into account when scheduling MapReduce jobs is very important in order to provide predictable services to users [33, 12]. To achieve QoS-aware scheduling using Quasar, however, users have to specify the various performance constraints of each of their workload applications. Using Quasar to schedule the workload applications used in this thesis will introduce additional and unnecessary complications of specifying the various performance constraints of the applications such as throughput and latency as well as quality of service requirements. The applications used in this thesis are long running and batch analytics applications where the load depends on the complexity of the algorithm and the size of the dataset to process. They do not require strict deadlines, therefore, do not reserve resources. The applications get executed whenever resource becomes available. In addition, Quasar is mostly suitable for applications hosted on public clouds such as Google Compute Engine [19], Amazon EC2 [2], Microsoft Azure [43], or large private clouds consisting of thousands of servers, where multiple applications (both low priority analytics and user-facing services that must meet strict deadlines) from different users and organizations need to share cluster resources in order to maximize performance and reduce cost of operation. This thesis utilized a private cluster, intended to evaluate long running and batch analytics applications and thus concurrent use of resources.

OPERA [66] is a more recent system similar to Quasar that uses profiles of runtime resource utilization as well as future resource availability to remove idle resources from active tasks and re-assign them to pending tasks for execution. The main motivation of this work is to solve the problem where resources are exclusively allocated to tasks and would only be released after the task has completely finished executing even though the task may not fully utilize all the resources allocated to it throughout its execution. According to the authors of OPERA, exclusive resource allocations leads to substantial underutilization of the cluster resources and degrading system performance.

However, not all tasks are suitable for opportunistic resource allocation (i.e. assigning occupied, but idle resources to tasks). The runtime architecture of OPERA classifies pending tasks into two categories: long and short running. It uses this information to determine the tasks that are eligible for opportunistic resource allocation. The design of OPERA only selects tasks that are identified as short running in order to mitigate the problem of severe resource contention - where tasks compete to access a shared resource, since long running tasks are more prone to this problem. To classify tasks, OPERA introduced additional complexity of using historic statistics of the same type of tasks in the same or other types of applications and frameworks to estimate the execution time of a task prior to actually executing. Therefore, OPERA is most suitable for applications that consist of multiple short running tasks. In contrast, most of the tasks in the applications used in this thesis are long running, therefore, OPERA may not provide substantial performance improvements for these applications. In addition, OPERA monitors tasks resource utilization at runtime in order to determine the available resources and dynamically re-assigns them to the waiting tasks. This dramatically increases the complexity of resource management system.

Omega [50] implements a novel scheduler architecture that uses a shared state approach and a lock-free optimistic concurrency control to achieve implementation extensibility and performance scalability. The

authors identify the two most widely used scheduler architectures - monolithic schedulers that use a centralized scheduling algorithm for all jobs, and two-level schedulers that employ an active cluster manager to offer resources to applications. Monolithic schedulers do not scale up well to very large clusters, and it is also difficult to add new requirements and features over time. This decreases efficiency and utilization, and also limits cluster growth. Similarly, the conservative resource visibility and locking algorithms used in two-level schedulers limits their flexibility and parallelism.

Borg [60] effectively runs applications across tens of thousands of machines. This cluster manager groups jobs into categories based on priority, e.g. high and low priority. When high priority tasks are not using all their reserved resources, the cluster manager can reclaim these resources and re-assign them to lower priority tasks. Borg is a highly reliable system which operates on the scale of clusters with tens of thousands of nodes. In addition, Borg runs the tasks based on priority, where applications are classified as high priority such as long-running server jobs, and low priority such as batch jobs. High priority jobs are allocated the highest share of the cluster resources. As an example, Google allocates 70% of the total CPU resources and 55% of the total memory resources to high priority jobs. In contrast, the testbed cluster used for the experiments in this thesis is small, and all the applications submitted to the cluster have the same priority. The only requirement is that applications should be executed when resources become available.

Both Omega and Borg are from Google, they are designed to be highly scalable and available. Therefore, they are most suitable for applications with high reliability and availability requirements including long-running services that should never go down and short-lived latency-sensitive requests (ranging from microseconds to milliseconds) such as the user-facing services at Google. As already mentioned earlier, the workload applications used in this thesis are analytical and batch in nature, without strict real-time latency requirements.

Schedulers

There have been a considerable number of studies in recent years devoted to improving the performance of job scheduling and resource allocation in Hadoop and Spark environments, with many researchers actively engaged in studies to better understand and improve the performance of schedulers and cluster managers.

Hadoop was initially designed to run batch workloads using the FIFO scheduler. As Hadoop became popular, however, and multiple applications from users needed to share the cluster resources, job response times started to suffer greatly due to the inefficiencies of the FIFO scheduler. Additional schedulers including Fair [23] and Capacity [22] were introduced to Hadoop in order to handle sharing of cluster resources. The idea behind the *Fair Scheduler* is to fairly share cluster resources among queues to ensure that applications get an equal number of resources on average. This scheduler offers several resource allocation policies, including memory-based fair sharing and multi-resource (e.g. CPU and memory) fair sharing using the Dominant Resource Fairness (DRF) [17] algorithm. On the other hand, the *Capacity Scheduler* allows multi-tenancy sharing of a large cluster between multiple organizations. This scheduler groups submitted applications into

queues, and provides each queue with a guaranteed share of the cluster capacity.

A key scheduling challenge in shared clusters is the ability to automatically adjust resource allocations at run-time to achieve performance goals for different applications. The previously mentioned schedulers do not efficiently handle applications with performance goals and Service Level Objectives (SLOs) such as application completion time deadlines. ARIA [59] addressed this problem by implementing a SLO-based scheduler in Hadoop that determines job ordering as well as the appropriate amount of resources to allocate to MapReduce jobs at run-time in order to meet service level agreement objectives. To incorporate SLO into job scheduling, ARIA builds a job profile for each job that gets executed regularly. The job profile contains the critical performance characteristics of the job during map and reduce stages. ARIA uses a MapReduce performance model that takes into account the job performance profile and the defined SLO to estimate the amount of resources required for the job to complete within the specified deadline.

Automatic or dynamic tuning and reconfiguration of Hadoop parameters has also been proposed in many studies as a viable means to improve the performance of MapReduce applications. For example, Starfish [24] introduced a self-tuning analytical tool to visualize and optimize Hadoop applications. Starfish uses a profiler which dynamically monitors jobs and collects profiles, and a sampler which collects data statistics to automatically adapt to workload and user needs in order to provide good performance. Although this approach is effective in some cases, it does not perform well for parameters that cannot be adjusted after a job starts, especially in a dynamic environment.

Although automatic tuning of MapReduce applications is appealing, it is difficult to select the best configuration parameter values for a single job running in either dedicated or shared clusters that would provide optimal performance. DynMR [54] offers a simple solution by interleaving the execution of partially completed reduce tasks and backfills map tasks to run in the same JVM in order to reduce resource underutilization during shuffle waiting times. The system uses a detection algorithm to identify underutilized resources during the shuffle phase and pro-actively schedules other waiting tasks to run on the identified resources through efficient context switching across different tasks.

Most existing MapReduce schedulers allocate a fixed amount of resources (e.g. CPU and memory) to jobs at the task-level, thus assuming the run-time resource consumption of the task is stable over its lifetime. Zhang *et al.* [71] argue that this does not provide optimal job performance since different tasks can have varying resource requirements during their lifetime. As reported in Herodotou *et al.* [24], MapReduce tasks can be divided into multiple phases of data transfer, processing and storage. Each phase has a distinct purpose and can be characterized by its uniform resource consumption over its duration. The authors propose a fine-grained phase-level resource-aware scheduler called PRISM which divides tasks (e.g. Map and Reduce tasks) into groups of phases where each phase has a similar and constant resource usage. PRISM improves job performance and resource utilization by allocating resources to phases within a job according to their resource requirements.

Other studies aim to improve Hadoop scheduling in terms of job completion times and data locality by

implementing new scheduling policies. For example, delay scheduling [67] aims to improve data locality (that is scheduling computation on the nodes that contain their input data) while maintaining fairness. Fair sharing of cluster resources between multiple users while preserving data locality (a very important feature of MapReduce that makes it efficient) is a challenging task in scheduler design, however, since the nodes that contain data for the job that is scheduled to run next according to fairness might not be available. In delay scheduling, jobs that should be scheduled next, according to fairness, delay their execution for a small amount of time in order to launch local tasks (that is on the nodes that contain their input data). Zaharia *et al.* [67] implemented delay scheduling into the Hadoop *Fair Scheduler*. This scheduler aims to maximize system throughput by providing fair sharing of resources using max-min fair sharing while retaining the performance provided by data locality. In addition, schedulers such as Quincy [30] were proposed to address the problem of scheduling with data locality and fairness constraints. The authors of Quincy model the fair scheduling problem as a graph data structure, where the edges weights of the graph represent various competing demands such as data locality and fairness. An optimal scheduling solution is computed using a standard graph solver according to a global cost model.

In data-intensive clusters, scheduling job execution on the same nodes where the jobs' data is stored is crucial for performance since transferring data across a network introduces additional network bottlenecks. In this type of environment, the requirements for fairness and data locality, however, often conflict. This is because a job scheduling strategy that is designed to achieve optimal data locality will typically delay a job until its ideal resources are available. On the other hand, fairness means ensuring that no single user or job monopolizes the entire cluster and delays the completion of other jobs. Fair scheduling means allocating available resources to a job as soon as possible, even if they are not the resources closest to the computation's data. Isard *et al.* formulated the fair-scheduling problem for cluster computing as a classic min-cost flow in a directed graph, and provided a graph-based framework for fine-grained cluster resource sharing with both fairness and locality constraints.

Interference between multiple virtual machines in a MapReduce cluster may cause performance degradation and negatively affect data locality-aware task scheduling policies such as delay scheduling. To address this, Bu *et al.* [9] proposed a task scheduling strategy to mitigate interference and virtualization overhead while preserving task data locality for MapReduce applications. Their solution includes an interference-aware scheduling policy, based on a task performance prediction model, and an adaptive delay scheduling algorithm for data locality improvement.

Most research work on Hadoop has been focused on understanding and improving the performance of MapReduce processing in terms of task scheduling optimization, adaptive resource provisioning and management, data locality improvement, application profiling as well as optimization and tuning of cluster parameters and test-bed configuration. For instance, Lin *et al.* [38] studied the impact of various Hadoop schedulers, including the capacity and fair schedulers, on job completion time. They evaluated their results using different combinations of scheduling policies such as configuring *Capacity Scheduler* with FIFO scheduling policy,

or *Fair Scheduler* with DRF scheduling policy, using three different scenarios: 1) one-queue cluster in which all applications are placed in one queue, 2) separated-queue where applications are placed in separate queues based on application type, and 3) merged-queue scenario where there are only two queues in the cluster. The results from this study shows that scheduling policies combinations suffer from problems such as resource fragmentation. Furthermore, the authors found that no single combination of the scheduling policies provides the best execution performance for all the three scenarios at all times. The merged-queue scenario, however, achieved the shortest turnaround time compared to the other two. This work mainly focused on evaluating the impact of the queue structure and of different combinations of scheduling policies in YARN under a mixed applications workload.

General Big Data Performance Measurement and Analysis

Liang *et al.* [36] used a standard benchmark to provide a performance comparison between Hadoop, Spark, and DataMPI in terms of system resource utilization including CPU utilization, memory utilization and network I/O throughput. The authors show that high performance communication mechanisms employed by DataMPI lead to better job execution and efficient resource utilizations compared to both Hadoop and Spark. This work analysed standard benchmark applications on three big data frameworks having different programming models. The authors fail to consider the job scheduling and resource allocation aspect of the systems. Furthermore, they mainly used standard benchmarks such as WordCount, Grep, Sort and K-Means for evaluations. Luckow *et al.* [40] investigate the applicability of deploying Hadoop on traditional High Performance Computing (HPC) platforms in order to support a class of scientific applications such as bio-molecular dynamics and genomics that require both the capabilities of high-performance computing environments and big data processing frameworks. The authors conclude that deploying Hadoop on HPC platforms allows diverse range of applications to take advantage of the various features provided by distributed systems. To achieve optimal performance, however, the Hadoop configuration parameters should be fine-tuned in order to utilize all the computing and storage power of the cluster.

Chaimov *et al.* [10] report on their experience in porting Spark to large production HPC systems. According to the authors, large scale HPC systems can perform slower than traditional workstation for distributed data processing frameworks like Spark due to the fact that HPC systems use global parallel file system such as Lustre [7], while workstations use local Solid State Drives (SSD) for data storage. Due to the presence of local disks in workstations, Spark shuffle performance is dominated by the network, while file system metadata performance dominates HPC systems. However, with techniques such as using a local file system and caching, the authors found that porting data processing frameworks to HPC systems can serve both scientific and data intensive workloads.

Understanding and improving the performance of in-memory frameworks compared to disk based systems has also gained considerable attention in recent years. Gu *et al.* [20] evaluated the performances of Hadoop and Spark in terms of time and memory cost when running iterative operations. Their results show that

Spark outperforms Hadoop in terms of speedup for this class of application, however, it comes at a cost of consuming more memory. Therefore, the speedup in Spark is trade-off for memory and can be badly affected where memory is insufficient to store all intermediate results, in which case Hadoop can have a better performance.

Awan *et al.* [5] perform an extensive performance characterization of Spark deployed on a single cloud server with non-uniform memory access (NUMA) architecture, and show that in-memory workloads do not scale linearly on single servers when the number of threads is high. The authors observed load imbalances between threads, additional CPU time spent by threads, and DRAM latency to be the major scalability bottlenecks in this deployment environment.

Ousterhout *et al.* [47] applied a methodology called blocked-time analysis that measures the amount of time a task spends blocked on a given resource to accurately identify and quantify Spark performance bottlenecks on SQL benchmarks. The authors identify CPU as the primary bottleneck and conclude that improving network performance will only slightly reduce job completion time.

Zhang *et al.* [72] compare the performance of Spark with a disk-based platform, SciDB [8] for large-scale matrix processing on scientific data. The authors observed linear speedup in Spark while caching intermediate data. However, Spark does not always outperform SciDB for these types of applications because caching intermediate results may increase pressure on memory, resulting in poor shuffle performance. Also, configuration parameters such as RDD caching, input data blocksize and memory compression can substantially influence job execution time. Although this research evaluated the performance of some of Spark's configuration parameters, it is not clear how these parameters will influence performance on different Spark supported cluster managers.

Although there are many cluster managers available, the work of this thesis only focuses on evaluating Spark applications on Standalone, Mesos and YARN. This is because the Standalone is provided as part of the Spark distribution, while Mesos and YARN are the most popular open source cluster management systems that support Spark applications out of the box. In addition, Mesos and YARN are advanced cluster managers designed to manage hundreds of computers and share cluster resources between tens of users executing multiple jobs concurrently using various scheduling algorithms and processing frameworks. These cluster managers are capable of managing the resources of the testbed cluster used for the purpose of the experiments.

Various schedulers including ARIA, PRISM, Starfish, DynMR, Delay Scheduling, Quincy etc., were discussed in this section. Most of these schedulers are only implemented and tested in Hadoop YARN. This thesis focused on evaluating the performance impact of the three cluster managers that supports Spark applications out of the box, above the level of schedulers. Therefore, the goal is to analyze performance results while tuning Spark configuration parameters in order to quantify the impact of the various parameters on image processing applications in terms of cluster resource utilization and makespan. The default schedulers provided by each of the three cluster managers tested (that is FIFO on Standalone, fair-sharing using DRF

algorithm on Mesos and *Capacity Scheduler* on YARN) were used for the experiments in this thesis. Studying the impact of various schedulers on the performance of distributed image processing applications on Spark is outside the scope of this thesis.

2.4.2 Image Processing on Distributed Systems

Recent advancements in image data acquisition methods and devices such as high resolution multispectral cameras are revolutionizing the way remotely sensed images are collected, managed and processed. Consequently, the volume, velocity, and variety of remotely sensed data generated and stored daily grows exponentially [41].

The rapid increase in the diversity and dimensionality of the remotely sensed data introduces substantial challenges in data storage, processing and interpretation. Typical image processing algorithms are traditionally resource (including CPU and memory) intensive. In addition, large collections of high resolution images occupy a large amount of hard disk space and routinely exceeds the memory size of a single computer. As such, extracting meaningful information from such large and high resolution data using image processing techniques is often difficult and time consuming on a single computer [37].

Examples of such representative image processing algorithms include contour detection [4], image registration [73], image splitting [63], image background correction, image segmentation and feature extraction [6]. Other remote sensing applications for earth monitoring and observation such as image classification [35], real time natural disaster monitoring [49] including earthquakes and floods, and earth observatory data processing [13] algorithms such as crop condition monitoring based on Normalized Difference Vegetation Index (NDVI) [65] requires fast and scalable processing of large image data with low latency. NDVI is a technique used to determine the density of green on a land by measuring and analyzing the difference between the wavelengths of visible and near-infrared sunlight reflected by the plants on the land. This simple technique is widely used by researchers to monitor plant health. Moreover, techniques such as remote sensing data fusion [70] have been developed and applied in various fields ranging from satellite earth observation to computer vision for integrating multiple data sources to produce more consistent, accurate, and useful information than that provided by the individual data source.

Therefore, there is a clear need for a more efficient platform for the storage and processing of remotely sensed images in a time and cost efficient manner [48]. Previous work has focused on utilizing high-performance computing (HPC) approaches and grid computing platforms (GCP) for the processing of such large high resolution image data. Despite the huge processing power of HPC systems and resource sharing architecture of GCP, there are still a number of challenges relating to efficient data storage and retrieval, network and I/O communication, runtime scalability and reliability. These challenges lead to a shift in the design of remote sensing systems from centralized environments towards distributed environments that allow accessing and processing of enormous quantities of data using a cluster of commodity computers [13].

Utilizing Hadoop MapReduce for efficient processing of large scale image data has gained attention from

both academia and industry in recent years. For example Chen *et al.* [11] proposed a high-performance cloud computing based Web Processing Service (WPS) framework for Earth Observation data processing using Hadoop as the underlying infrastructure for supporting the cloud computing environment. The authors of the paper combined the flexibility and extensibility provided by WPS with the performance, scalability and efficiency of Apache Hadoop to evaluate the feasibility of implementing a Hadoop based WPS framework in cloud environment for the processing of large scale image data.

In a similar work, Lin *et al.* [37] designed and implemented a cloud computing platform based on Web Coverage Service and Web Map Service interfaces from the Open Geospatial Consortium (OGC).[11] The proposed architecture used the Hadoop Distributed File System for data storage, and Hadoop MapReduce for distributed image processing. The performance of the new system was evaluated through extensive read/write experimentation using four types of data sets with different read/write access distributions including normal distribution, skew to left, skew to right, and peak in left and right. According to the authors, write/read performance with HDFS on the proposed system outperformed a local file system for large files.

A more recent work [18] designed an extensible and adaptable geospatial data processing framework based on Hadoop's implementation of the MapReduce programming model to enable the management and processing of spatial and remote sensing data in distributed environments. The aim of the new framework is to enable easy adaptation and porting of algorithms and applications already implemented on other systems into the Hadoop distributed framework with minimum effort from the programmer's part. Similarly, Yan *et al.* [64] designed a large scale image processing research cloud based on Hadoop, and evaluated the performance and scalability of the system using three widely used image processing algorithms including Discrete Fourier Transform (DFT), face detection, and template matching.

The above mentioned works on distributed image processing on Hadoop focused on how the existing MapReduce programming model and HDFS can be adapted and used for distributed processing and storage of large scale image data. Despite the number of works that were successfully done in this area, however, it is challenging to port the Hadoop system for image processing due to highly technical complexities of the Hadoop system architecture. *Hadoop Image Processing Interface (HIPI)* [53] aim to simplify the development of image processing and computer vision applications by providing users with highly flexible and easy to use interface for implementing complex image processing and computer vision algorithms using the MapReduce programming model. The framework hides the highly technical details of the Hadoop MapReduce framework, and removes the complexity of writing distributed image processing applications by providing a high-level image library with access to large-scale image data and computation resources of a distributed system.

Similar to HIPI, *Hadoop Image Processing Framework* [57] provides a Hadoop-based library which allows software developers to write large scale image processing applications on the Hadoop MapReduce framework without having to master the technical complexities of the framework. As such, researchers and programmers

[11] Open Geospatial Consortium is an international organization that provide free and open standards for the global geospatial community through consensus.

with minimum skills in developing distributed applications that use large image data could focus on implementing their image processing algorithms instead of worrying about the architecture and implementation of the underlying Hadoop framework.

Moreover, there is a lack of standard image processing benchmarks and stress tests for evaluating the performance of Hadoop on image processing applications. Most previous Hadoop benchmarks primarily focused on evaluating the performance of Hadoop in processing textual data which are different from high-dimensional images with large file size. To fill this gap, Bajcsy *et al.* [6] present the characteristics of four representative image processing applications (consisting of both computation intensive and data intensive) on a Hadoop cluster. The authors used Terabyte size image data to evaluate the performance of image processing applications on Hadoop in terms of computational scalability and compared their results against standard Hadoop benchmarks such as the Terasort and Teragen.

High performance queries on large volumes of spatial data are essential in many applications. However, the high computational requirements of performing complex queries such as cross-matching (that is large scale spatial join) on massive volumes of spatial data posed substantial challenges in geodatabase management. Aji *et al.* [1] implemented a scalable and high performance spatial data warehousing system for running large scale spatial queries on Hadoop. This new library called *Hadoop-GIS* is integrated into Hive [55] and utilizes the MapReduce programming model to provide support for performing multiple types of spatial queries on large datasets using a Hadoop cluster of commodity computers. Apache Hive is a data warehouse system for querying and analysing large datasets using an SQL-like interface on Hadoop. The authors of *Hadoop-GIS* have shown through experimentation that the new system achieved better performance than traditional Spatial Database Management Systems (SDBMS) for compute intensive queries in terms of query response time and scalability.

SpatialHadoop [16] is another framework that extends Hadoop to support fast and efficient processing of large spatial data using the traditional MapReduce programming model. It improves the performance of MapReduce applications that process spatial data by providing efficient spatial data storage, indexing, and spatial query support to Hadoop. The framework provides a high-level SQL-like language for spatial data operations by extending the Pig Latin [46] language. Pig Latin is a simple SQL-like scripting language that is used with Apache Hadoop for analysing large datasets.

Other researchers [61] compare the performance of running different image processing algorithms using different big data computing paradigms including MPI and MapReduce with varying computational complexity and data size.

Many image processing and remote sensing applications run a set of computations multiple times on the same subset of data such as multi-iteration singular value decomposition (SVD) feature extraction algorithm; thus MapReduce based batch processing is not efficient, because MapReduce shuffles intermediate data to disk during each iteration. For such applications that reuse intermediate data multiple times, disk and network I/O becomes the bottleneck. In order to address this, researchers have shifted attention towards

the utilization of the in-memory abstraction of Spark for fast and efficient processing and analysis of massive remotely sensed data.

For example, Huang *et al.* [27] utilized a Spark cluster running on top of Hadoop YARN (as the cluster manager) to facilitate the design and implementation of generic algorithms on BigData platform for parallel processing of massive Remote Sensed data. Spark eliminated the problems introduced by Hadoop through loading intermediate data into a distributed memory and re-using the loaded data multiple times throughout the lifetime of an application without relying on rather slow disk access for input and output of intermediate data. Using this model for multi-iteration image processing algorithms, Spark based processing can provide an order of magnitude speed-up compared to Hadoop MapReduce based algorithms.

Similarly, Sun *et al.* [52] present a Spark based platform for processing of massive remote sensing data. The platform uses Spark for distributed in-memory processing, HDFS for distributed storage of massive datasets and Hadoop YARN for job scheduling and resource management. The authors show that Spark is often one order of magnitude faster than Hadoop for executing the singular value decomposition (SVD) algorithm.

The above mentioned previous studies related to image processing on distributed systems mainly present a Hadoop based high performance cluster for distributed processing of large scale image data in the cloud. Most of the proposed architectures utilized the Hadoop MapReduce programming model for implementing image processing and remote sensing algorithms, and HDFS for efficient and reliable storage of massive image data. The few studies that used Spark as the underlying engine for image processing only evaluate the performance gain (in terms of running time) of generic algorithms running on the Spark platform using Hadoop YARN as the cluster manager, and compare the performance of the in-memory based Spark with batch oriented Hadoop.

This thesis focuses on evaluating the performance impacts of running multiple heterogeneous image processing applications on different testbeds including a large server, and the three Spark supported cluster managers i.e. Standalone, Mesos and Hadoop YARN. The two different job scheduling and resource allocations modes available on the three cluster managers i.e. static and dynamic resource allocation were studied. In addition, the experiments explore the various configurations that control speculative execution of tasks, resource size per executor, and the number of parallel tasks per job, and what effects they have on image processing applications.

Furthermore, the work in this thesis focused on using three domain specific image processing applications (instead of the generic big data benchmark applications) including Image Registration [73], Flower Counter and Image Clustering to provide an extensive analysis of the impact of various Spark configuration parameters on the performance of these applications in terms of cluster resource utilization and makespan. The main standard benchmark applications available in the literature for Hadoop MapReduce includes Sort, TeraSort, WordCount, and Grep [26], while the popular benchmarks for Spark includes PageRank, SVD++, TriangleCount, RDDRelation, PageView, Logistic Regression and Matrix Factorization [34]. These applications

mainly process text data, and are not suitable for our use case and deployment target of processing binary data (images) which are substantially different from line-by-line text processing.

2.5 Summary

This chapter presents a detailed description of the background and previous related works in the areas of job scheduling, resource allocation and image processing on distributed systems. The definitions of the various terms and concepts used in this thesis, as well as a summary of literature survey were presented. In addition, the chapter provides an overview of the Apache Spark architecture.

Spark applications are divided at runtime into *Jobs, Stages,* and *Tasks*. These applications consist of various components including *Driver*, *Executors* and *SparkContext*, and can connect and request resources from various cluster managers including Spark Standalone, Apache Mesos, and Hadoop YARN. Furthermore, Spark applications can access data stored in local or distributed filesystem such as HDFS. Spark implements two different resource scheduling and allocation policies: static and dynamic, and provides many configuration parameters to control the runtime behaviour of the Spark system.

Chapter 3

Experimental Design

This chapter presents the experimental design methodology used in this thesis. The first part of the chapter gives a detailed description the tools used in the experiments, including the hardware configurations of both the testbed server and cluster, as well as their software specifications. The second part of the chapter describes the configuration parameters used and how they were set for the different experiments to evaluate the performance impact of (Spark's job scheduling and resource allocation) configuration parameters on image processing applications. A description of the three heterogeneous workload applications used in this thesis, as well as the performance metrics selected for their evaluation are also presented. The last part of the chapter presents the description of the multiple experiments done.

3.1 Hardware and Software Configurations

This section describes the hardware setup as well the software configurations of the two testbeds used in the different experiments of this thesis. Information about operating system, processor, memory, disk, and number of cluster nodes is provided. All the nodes of the Spark cluster and the testbed server run Ubuntu 16.04.3 LTS Operating System.

3.1.1 Sever Configurations

The server machine *onomi.usask.ca* (called *Onomi* in the rest of this document) was used as a testbed for running the sequential versions of the three image processing applications. The computer is configured with two Intel® Xeon® E5-2690 v4 CPUs (14 cores @ 2.6 GHz on each CPU, hyper-threaded, for a total of 56 virtual cores), 620 GB RAM and 43 TB of disk space. This machine runs Python version 3.5.2.

Exclusive access to *Onomi* was provided for a limited time frame to run the sequential version of the three image processing applications and evaluate their performance on a single large computer.

3.1.2 Spark Cluster Configurations

The DISCUS Lab at the University of Saskatchewan has a heterogeneous Spark cluster consisting of 12 virtual machines on 12 physical machines, with 1 master node and 11 worker nodes. The nodes are interconnected

using a 1 Gigabit private network. This cluster was used as a testbed for running the distributed version of the image processing applications.

The master node and each of the first nine worker nodes are configured with a single Intel® Core™ i7-2600 CPU (4 cores @ 3.4 GHz, hyper-threaded, for a total of 8 virtual cores). In addition, the master node is equipped with 11 GB RAM and 396 GB of disk space, while the nine worker nodes are equipped with 14 GB RAM and 7.4 TB of disk space. Each of these worker nodes is configured to provide the cluster with 7 virtual cores and 13 GB of memory. The remaining two worker nodes have slightly different configurations. They are each equipped with a single Intel® Xeon® E5-2403 CPUs (4 cores @ 1.8 GHz, hyper-threaded, for a total of 8 virtual cores), 35 GB RAM and 7.7 TB of disk space, and provide the cluster with 7 virtual cores and 34 GB of memory.

With these configurations, 1 virtual core and 1 GB memory were reserved for the Operating System and other background processes on each of the 11 worker nodes. The cluster runs Spark version 2.1.0, Hadoop YARN version 2.7.2, Mesos version 1.2.0, and Python version 2.7.12. Table 3.1 shows the hardware configurations of the two testbeds. In all, the server has a total resource of 56 virtual cores and 620 GB memory, while the cluster has a total resource of 77 virtual cores and 185 GB memory for scheduling and allocating to different Spark applications by the cluster managers as shown in Table 3.2.

The heterogeneity of the testbed cluster comes from the fact that two of the worker machines have slower CPU cores than the rest (approximately half the speed). Experiments show that tasks scheduled on these slower machines sometimes tend to take longer time to finish than the rest. To overcome the effect of having slower machines slow down the processing of an application, speculative execution of tasks was evaluated to verify its impact on the makespan of the set of jobs. With speculative execution of tasks enabled, the tasks that are running slowly in a given stage may be re-launched on a different executor.

Table 3.1: Testbeds Hardware Configurations

Node	CPU	Memory	Disk
Master Node	Intel® Core™ i7-2600 @ 3.4 GHz	11 GB	396 GB
9 Worker Nodes	Intel® Core™ i7-2600 @ 3.4 GHz	14 GB	7.4 TB
2 Worker Nodes	Intel® Xeon® E5-2403 @ 1.8 GHz	35 GB	7.7 TB
Onomi	2 Intel® Xeon® E5-2690 v4 @ 2.6 GHz	620 GB	43 TB

3.2 Chosen Parameters

In addition to supporting the 3 chosen cluster managers (Standalone, Mesos, YARN), and two different modes of resource allocation i.e. static and dynamic resource allocation, Spark provides many configuration parameters that control speculative execution of tasks, size of resources per executor, and the number of parallel tasks that can be created and executed for a giving job. This section provides a detailed description

Table 3.2: Cluster Resources

Node	Virtual Cores		Memory		Disk	
	Node Total	Cluster	Node Total	Cluster	Node Total	Cluster
9 Worker Nodes	8	7	14 GB	13 GB	7.4 TB	7.4 TB
2 Worker Nodes	8	7	35 GB	34 GB	7.7 TB	7.7 TB
Totals	88	77	196 GB	185 GB	82 TB	82 TB

of the parameters chosen for the various set of experiments in this thesis, and how they are configured for comparison between running Spark applications on the three supported cluster managers.

3.2.1 Parameters to Control Resource Scheduling and Allocation

There are several parameters that control how static and dynamic resource allocation behaves. However, for this thesis only two parameters were used, namely *spark.dynamicAllocation.enabled*, which indicate whether to use static or dynamic resource allocation, and *spark.shuffle.service.enabled*, which allows safe removal of executors from applications without losing their shuffle data. Both of these parameters need to be set to *true* in order for Spark applications to use dynamic resource allocation.

3.2.2 Parameters to Control Number of Parallel Tasks Per Job

Spark provides several options for controlling the number of RDD partitions (and concurrent tasks) that can be created at any given stage. For applicatioins with stages that involve distributed shuffle operations, such as *reduceByKey* and *join*, the *spark.default.parallelism* configuration parameter can be used to control the number of partitions in RDDs returned by these operations.

In addition, Spark also provides fine grain control over the number of partitions at run time. By changing the *numPartitions* variable in Spark transformations such as *repartition*, *coalesce*, and *repartitionAndSortWithinPartition*, the number of partitions in an RDD at any stage of a job can be controlled. Furthermore, the *spark.files.maxPartitionBytes* configuration parameter can be used to define the maximum number of bytes to pack into a single partition when reading files, which will also control the number of RDD partitions. Since Spark creates one task per RDD partition, these parameters control the total number of tasks at any stage of an application. For the experiments in this thesis, *spark.files.maxPartitionBytes* was used to control the size of Spark RDD partitions (and the number of concurrent tasks) in the three benchmark applications.

3.2.3 Parameters to Control Speculative Execution of Tasks

Speculative execution of tasks is a feature of Spark that enables the detection of slow running tasks and rescheduling their execution on different executors in order to prevent having failed or slow machines slowing

down the execution of an entire application. Slow running tasks are considered to be tasks running slower in a stage than the median of all successfully completed tasks in that stage according to configuration values. If a node that is running a task crashes, Spark will automatically re-run the task on another node; if the node is still active but slower than other nodes, however, Spark will not re-launch the task on another node by default.

With *spark.speculation* set to *true*, slow running tasks in a stage may be re-launched on another node, and Spark would use the results of the task that finishes first. There are various parameters associated with speculative execution in Spark. These parameters include *spark.speculation.interval* which controls how often Spark checks for tasks to speculate, *spark.speculation.multiplier* which is used to control how many times slower a task is than the median to be considered for speculation, and *spark.speculation.quantile* which controls the fraction of tasks that must be complete before speculation is enabled for a particular stage. For the experiments in this thesis, only the *spark.speculation* configuration parameter was used to control speculative execution, all other parameters associated with speculation are set to default values.

3.2.4 Parameters to Control Resource Size Per Spark Executor

Two Spark configuration parameters, *spark.executor.cores* and *spark.executor.memory* control the number of CPU cores and memory per Spark executor process respectively. The value for *spark.executor.cores* is set by default to 1 in YARN mode, and all available cores on the worker node in Standalone and Mesos modes. The value of *spark.executor.memory* is set to 1 GB in all the three cluster managers by default.

Using the default values for these parameters means that only one executor per worker node (with 1 GB memory and all available CPU cores on the worker machine) are allocated to Spark applications by the Standalone and Mesos cluster managers. On the other hand, the YARN cluster manager would allocate multiple executors per worker nodes, however, the executors would use the minimum possible resource (that is 1 Core and 1 GB memory). Setting these parameters allows one to control the amount of resource per executor. This means multiple executors with appropriate resource size may run on a single worker node.

3.3 Performance Metrics

Although there are many metrics that can be used to evaluate the performance impact of different Spark configuration parameters on image processing applications, the work in this thesis use job makespan and cluster CPU and memory utilization for evaluations. This is because makespan is a very useful metric to understand the overall throughput of a system, it can be used to effectively quantify how changing the values of various configuration parameters affect the overall system performance. In addition, the goal of the thesis is to parallelize the three image processing workload applications to determine if they are suitable for running in a distributed Spark environment simultaneously compared to simply running them on a large single computer. Since there are many applications that need to run concurrently and share resources, it is

important to compare the time taken to process a set of jobs (i.e. from the submission of the first job to the completion of the last job) under different parameter values. This is the *makespan*.

In addition to makespan, resource utilization is another key indicator of an application's performance. Resource utilization tracks how busy various resources of a computer system are when running an application. To evaluate the performance of the three Spark cluster managers (i.e. Standalone, Mesos and YARN) in terms of their ability to efficiently schedule jobs with various resource requirements to take advantage of available resources and minimize total makespan, two other important performance metrics were used, the cluster CPU and memory utilization. These performance measurements can help determine how various parameter tuning has affected the overall performance of Spark. Furthermore, on few occasions, individual job waiting and execution times were used to help in interpretation and analysis of results. The description of the various chosen performance metrics is provided below:

1. Job Makespan: This is the total time that elapses from submitting the first job until the end of the job that finishes last. The makespan includes both the waiting and processing time of all jobs.

2. CPU utilization: The CPU utilization of a computer system is the usage over time of the system's processing resources (that is the amount of work handled by the CPU). In this thesis, the CPU utilization refers to the percentage of CPU resource of all nodes in the cluster used over time for processing all submitted jobs.

3. Memory utilization: The memory utilization refers to the fraction of the memory resource of all nodes in the cluster used over time for processing all submitted jobs.

4. Job waiting time: This is the time a job spends between submission and getting the first resource allocation (that is the first Spark executor) from the cluster manager.

5. Job execution time: This is the time a job spends executing its tasks on the cluster. It does not include the waiting time.

To get the makespan values, the jobs submission scripts discussed in Section 3.6.1 record the beginning and the end time of each experimental run. Using these values, the total makespan was calculated. In addition, the Spark web UI provide various information about running jobs, including executors allocated or removed, waiting and execution times of jobs, etc. Spark also logs these data to disk for later analysis. The log file generated by Spark for each job was used to get the job waiting and execution times.

Furthermore, an external monitoring system called Ganglia [42] was used to get the cluster CPU and memory utilization of running applications. Ganglia is a scalable distributed monitoring system for high-performance computing systems such as clusters. The system could be used to view either live or recorded statistics covering metrics such as CPU load averages, memory, disk I/O, network utilization etc. for many nodes in a cluster. Ganglia version 3.6.0 was used for the experiments. Analyzing the logs generated by Spark, and cluster monitoring logs generated by Ganglia give an insight as to how different Spark cluster managers

schedule and allocate resources to jobs, and how these resource allocation behaviours affect applications execution times and cluster resource utilization.

Other performance metrics such as network utilization, disk I/O, number of executors allocated could be used to measure and evaluate the performance of Spark. However, due to time limitations, these metrics are not considered in this thesis.

3.4 Description of Datasets

There are two categories of images used for the experiments in this thesis based on the method of collection, including *Drone Images* and *Still Camera Images*. Each experimental run involves processing the nine different datasets shown in Tables 3.3 and 3.4, having a total size of 9.7 GB and 15861 images in total. Although the sizes of the individual datasets are relatively small with respect to Big Data, the datasets contain binary files (images) which requires a considerable amount of time to process compared to text files. Furthermore, unlike the generic Big Data workloads that process text files, the image processing applications used for the experiments in this thesis require a substantial amount of time to run. Nevertheless, the multiple workloads used to evaluate Spark configuration parameters are enough to stress the 12-node (77 cores) testbed cluster. For example, with 128 MB Spark RDD partition size, the 9.7 GB dataset would be divided into approximately 77 tasks during each experimental run, with each task having approximately 206 images to process. Reducing the RDD partition size further would increase the number of tasks to process by each of the CPU cores.

Drone Images

These are aerial images of canola test plots captured using a multispectral camera fixed on a drone. The camera captures images in five bands including Blue, Green, Red, Red Edge and Near-Infrared. In addition, the five bands are captured in Grayscale colorspace and stored in two different file formats: Portable Network Graphics (png) and Tagged Image File Format (tiff). The .png versions of the multispectral images are used for the Image Registration application. Due to a slight variation in the number of image sets captured during different drone flights, the first 200 image sets from each drone images folder were used as dataset for the Image Registration experiments. Table 3.3 shows the characteristics of the drone images used for various instances of the Image Registration jobs. The "Dataset Name" column in the table represents the date when the images were captured (in the format DDMMYYYY) followed by the format of the images used. For example the dataset *23062016_png* consist of *.png* images captured on *June 23, 2016*.

Still Camera Images

These are overhead images of plots captured using cameras fixed at various locations in the same set of test plots. The images are captured in standard Red Green Blue (sRGB) colorspace and stored in Joint

Table 3.3: Drone Images Characteristics

Job Name	Dataset Name	Dataset Size	Number of Images	Date Captured
imageRegistration_job_1	23062016_png	1.7 GB	1000	June 23, 2016
imageRegistration_job_2	14072016_png	1.7 GB	1000	July 14, 2016
imageRegistration_job_3	06082016_png	1.7 GB	1000	August 06, 2016

Photographic Experts Group (JPEG) file format. These images are used for the various instances of the Flower Counting and Image Clustering applications. Table 3.4 shows the characteristics of the still camera image datasets. The "Dataset Name" column in the table represent the date when the images were collected (in the format DDMMYYYY) followed by the ID of the camera that captured the images, and a name representing the actual date the images were captured. For example the dataset *15072016_1108_images12* consist of images captured by camera *1108* on *July 12, 2016* and collected on *July 15, 2016*.

Table 3.4: Still Camera Images Characteristics

Job Name	Dataset Name	Dataset Size	Number of Images	Date Captured
flowerCounter_job_1	15072016_1108_images12	314 MB	1020	July 12, 2016
flowerCounter_job_2	15072016_1108_images14	354 MB	1020	July 14, 2016
flowerCounter_job_3	15082016_1108_images0	422 MB	719	August 2, 2016
imageClustering_job_1	15072016_1108_images1_7_10_13	1.3 GB	4080	July 1, 7, 10, 13, 2016
imageClustering_job_2	15082016_1108_images1_7_10	1.4 GB	2962	August 3, 9, 12, 2016
imageClustering_job_3	15072016_1108_images2_4_5	850 MB	3060	July 2, 4, 5, 2016

3.5 Workload Applications

This section presents a detailed description of the three heterogeneous workload applications used in this thesis, including Image Registration, Flower Counter, and Image Clustering. The Image Registration and Flower Counter applications were developed at the Department of Computer Science, University of Saskatchewan, while the Image Clustering application was adapted from a version obtained from the Internet.[1] Furthermore, the Image Registration and Image Clustering use standard image processing methods that are available in the literature. In this thesis, an application refers to the implementation of the three image processing applications used for evaluation, while a job refers to an instance of any of the three applications.

The heterogeneity of the three workload applications comes from the fact that each of the application

[1] `http://scikit-learn.org/stable/auto_examples/cluster/plot_kmeans_digits.html` (28 Jul, 2018)

has a different resource requirements in terms of CPU and memory. Specifically, the Image Registration application is very CPU intensive, because it involves a lot of matrix transformation and arithmetic, while the Image Clustering application is CPU intensive during K-means iteration stage and I/O intensive during clustering stage. The Flower Counter application has a medium requirement for both CPU and memory when compared with the other two applications. Table 3.5 gives a summary of the various characteristics of the three workload applications. The "Max CPU utilization (%)" row in the Table shows the maximum percentage of CPU used by each of the applications when run in isolation on *Onomi* with 56 cores (for the sequential applications), and Spark with 77 cores (for the distributed applications). Similarly, the "Max memory utilization (%)" row shows the maximum percentage of memory used by each of the application on *Onomi* with 620 GB of memory, and Spark with 185 GB of memory.

Table 3.5: Workload Applications Characteristics

Applications	**Image Registration**		**Flower Counter**		**Image Clustering**	
	Sequential	Distributed	Sequential	Distributed	Sequential	Distributed
Lines of code	436	355	789	766	156	124
OpenCV functions	5	9	14	14	2	2
Spark functions	-	8	-	31	-	15
Spark jobs	-	2	-	31	-	8
Spark stages	-	3	-	54	-	14
Max CPU utilization (%)	99	99	24	26	11	57
Max memory utilization (%)	0.5	54	0.9	65	9	66

3.5.1 Sequential and Distributed Versions

The three workload applications mentioned above were initially developed for sequential processing using Python. These are referred to as the sequential versions. Due to the large number of images to process and the high resource requirement of the various image processing algorithms used in developing the applications, the processing takes a long time to complete even on a large server equipped with many resources. This brings the need to find better ways to efficiently speed up processing.

Therefore, it is important to parallelize and evaluate performance of the already written Python code to see if the applications are properly amenable to distributed processing using the Spark platform. With this in mind, I converted the sequential versions of the workload applications to run on Spark. These are referred to as the distributed versions. Various Spark transformations and actions such as *map*, *filter*, *aggregate*, as well as optimized and distributed machine learning algorithms such as *K-Means* clustering were used to replace the sequential versions provided by Python.

3.5.2 Image Registration

Remotely sensed data continues to grow in volume and diversity. Images are one of the most important categories of such data, providing large amounts of data representing multiple time or simultaneous observations of the same features by different sensors. The integration of information from multiple sources, such as images from different sensors starts with the registration of the data into single spatial coordinate system. Data may consist of multiple photographs, data from different sensors, times, depths, or viewpoints. Image registration is widely used in computer vision, and in compiling and analysing image data from satellites.

To register multiple images on the same coordinate system, one of the images is selected as the reference image and the others are respectively referred to as the target images. Image registration involves spatially registering the target image(s) to align with the reference image. There are multiple methods of registering images; the image registration application developed for the experiments in this thesis uses a feature-based method to register images. In this method, features such as points, lines, and contours are used to find correspondence between multiple images.

Each instance of the Image Registration application takes in a single directory containing drone images in a *.png* format for a single drone flight. The size of the input directory depends on the number of images collected during the drone flight; nevertheless, each drone flight (that is a single directory of images) creates at least one image registration job. The images are in a groups of 5, representing 5 different channels captured by the drone. Therefore, each image registration job consist of many groups of images representing different coordinates in the farmland, with each group containing 5 images of approximately the same coordinates. Due to the image capture mechanics on the drone camera,[2] the 5 images in a group are not aligned on the same coordinates, which makes it impossible for any image processing to be done on these images without first being registered.

The sequential version of this application reads the input images from a local file system using Python's *skimage.io* module and registers multiple channels of images using the OpenCV's *cv2.xfeatures2d.SIFT* class. This class uses the Scale Invariant Feature Transform (SIFT) algorithm [39] to extract images features and computes descriptors of the extracted features. These features are passed to a features matching function which uses the OpenCV's *Brute-Force Matcher* to match the descriptors of the features. The *Brute-Force Matcher* simply takes the descriptor of one feature in the reference image and matches it with all other features in the target images using some distance calculation. The descriptor with the closest distance is returned. The *knnMatch* method of OpenCV's *Brute-Force Matcher* is used to return k best matches where k is specified by the user. Finally, *cv2.findHomography* function is used to find the perspective transformations between the two planes detected.

For each group of 5 input images, the application tries to align them together on a single coordinate system. The outputs are a set of 5 registered images, a coloured RGB image produced by adding channel 1,

[2] MicaSense RedEdge-M multispectral camera was used to capture the images. This camera has five different spectral bands (blue, green, red, red edge, near-IR) which capture images at slightly different time and not perfectly aligned.

2 and 3, a cropped version of the RGB image produced by removing a black border surrounding the RGB image, and NDVI image of the coordinates. The output images are saved on the local file system.

This process of finding features and descriptors and matching the detected features across all the images in the same group is CPU intensive and takes long time to finish for each group. Also, experiments show that the amount of data within the images in a group affect the processing time (that is the larger the data within the images, the more time the processing will take).

From the discussion of the sequential version of Image Registration application above, it is clear that this application is embarrassingly parallel since each group of 5 images can be processed separately and independent of processing of other groups. With this in mind, I re-write the sequential version of the Image Registration application for distributed processing on Spark.

All processing on the image groups is similar as with the sequential version. However, the distributed version of the application uses Spark's *binaryFiles* to read data from HDFS (which is a distributed file system). The application then creates an RDD of all the images for a single Image Registration job. Each image file is read as a single record of key-value pair, where the key is the path of the file, and the value is the content. A *map* transformation is used to create a new RDD of the images converted from binary files into Python's numpy array of image data. The *reduceByKey* transformation is used to group all the images belonging the same group together. Spark's *foreach* operation is invoked on the created RDD. This action applies the feature detection and descriptor matching function on each group (that is Spark RDD partition) in parallel on different Spark executors. Each RDD partition in this case contain a group of 5 images, together with their names and group key. The output of the distributed version of the Image Registration application is saved on a Network File System (NFS) available on all nodes. This file system is mounted to HDFS, which makes all the files stored on HDFS accessible from NFS. This application is very CPU intensive, because it involves a lot of matrix transformation to find features within the images to register. It also uses matrix arithmetic to produce both RGB and NDVI versions of the images.

3.5.3 Flower Counter

The Flower Counter application processes images captured by cameras positioned at various locations in a farmland by using various image processing techniques to count the approximate number of Canola flowers in the image.

The sequential version of the application reads in multiple images from the local file system or from a database lookup. An object of *CanolaTimelapseImage* class is created for each image read. This class consists of many methods corresponding to different processing, including *readImage* for reading the image data, and *showDetectedBlobs* for highlighting the detected flowers. Images are read and converted into RGB using Python Imaging Library (PIL). A pre-processing step is done on the input images to filter those that are likely to impact flower estimation results negatively, for example images captured at night and images

captured on a rainy day. To filter the images, *sklearn*[3] implementation of *K-Means* clustering algorithm is used. *K-means* is a widely used clustering algorithm for knowledge discovery and data mining. It aims to partition input data objects into k clusters by calculating the nearest mean cluster to which an object belongs.

The filtering stage starts by defining four coordinates representing a square boundary specifying the region of interest within the image where the flowers would be counted. The next stage computes histograms of each individual input image (pixel) data in both greyscale and CIELAB colour space. Each pixel in the input image is grouped into one of 256 bins. All the input images are passed into a method that computes histogram shifts for each image with respect to the average histograms of Lab colour space. This computation involves selecting the first image histogram as a reference and computing the discrete cross-correlation between the reference image histogram and the histograms of all other input images. Furthermore, a correlation reference is computed by cross-correlating the reference histogram and the average histograms of all the input images. The correlation value of each input image plus the correlation reference computed is returned as the histogram shift of that image. Images are flagged as good or corrupted based on the value of their greyscale histogram value.

The images that are flagged as not corrupted are passed to the *K-Means* clustering stage. This stage groups all the input images into 2 clusters (namely cluster 0 and cluster 1) based on the percentage of yellow pixels within the image plot boundary. The images in cluster 0 are the most suitable for the flower counting algorithm and thus used in the next stages of the flower counting pipeline. The selected images are sorted from high to low based on the percentage of yellow pixels. Thus, the image with the highest number of yellow pixels is the first on the list. This sorted list of images is passed to the flower counting method which runs a pipeline of various image processing techniques to estimate the number of Canola flowers in each input image. The processing pipeline includes converting the images from RGB to CIELAB colour space, computing sigmoid mapping, and finally detecting blobs within the defined image region using scikit-image Determinant of Hessian (DoH) blob detection algorithm. For each blob found, the blob detection method returns its coordinates and the standard deviation of the Gaussian kernel that detected the blob. The number of flowers were estimated by counting the number of the returned blobs. The output of the Flower Counting application is a text file containing image names and estimated number of Canola flowers per image. This text file is written to a local file system for analysis.

Due to the large number of images, and the various processing involved in this application, it takes a large amount of time to complete. However, unlike the Image Registration application which is embarrassingly parallel, the stages in the Flower Counter can be categorised into two classes: 1) Stages that process a single image independent of other images such as the final flower counting stage, and 2) Stages that require information from all other input images such as *K-Means* clustering stage.

To distribute the processing of the Flower Counter application on multiple computing nodes, I re-write

[3] *sklearn* is a Python module that implements several machine learning algorithms

the sequential version of this application, which uses pure Python, into a distributed Spark application using *pyspark's* transformations and actions. Similar to the sequential version, the distributed version of the application uses the same set of image processing pipelines. The main difference is that the Spark version of the application reads input images in a .jpg format from HDFS. Spark's *binaryFiles* is used to create an RDD of all the images for a single Flower Counter job. Each image file is read as a single record of a key-value pair, where the key is the path of the file, and the value is the content. A *map* transformation is used to create a new RDD of the images converted from binary files into Python's numpy array of image data.

Spark provides many functions that can process RDD partitions in parallel on multiple executors. One of such function is the *aggregate* transformation. It aggregates the elements of each RDD partition, and then the results for all the partitions, using a given *combine function*. This transformation was used to calculate the average histogram of pixels over all input images in parallel. Other Spark transformations and actions including *map*, *filter*, *zipWithIndex*, *sortBy*, *coalesce* and *saveAsTextFile* were used for executing various processing in parallel. For the *K-Means* clustering stage, Spark's machine learning library *MLlib* provides an efficient implementation of the *K-Means* clustering algorithm that includes a parallelized variant of k-means++. This implementation of *K-Means* performs processing in parallel; therefore, it was used to replace the *sklearn* implementation in the Spark version of the Flower Counter application. The output of the final stage (that is the flower counting stage) is reduced into 1 RDD partition using *coalesce*, and saved to HDFS as a text file containing the image file name and the estimate of Canola flowers count per image. This final stage involves shuffling output data over the network.

3.5.4 Image Clustering

Clustering is an unsupervised learning task of grouping data such that members of the same group (or cluster) are more similar to each other by a given metric than they are to the members of the other clusters [21]. The Image Clustering application used in this thesis aims to cluster image data captured by cameras positioned at various locations in the plots mentioned previously into three clusters based on image features.

Similar to the other two workload applications discussed above, the sequential version of this application reads in multiple images from the local file system, and uses the *sklearn* implementation of *K-Means* clustering to cluster the input images using random initialization mode with a maximum of five iterations. The output of the Image Clustering application is a text file containing images name and the cluster number to which they belong. This text file is written to the local file system for analysis.

The sequential version of the Image Clustering application was rewritten for distributed/parallel processing using Spark's implementation of *K-Means*. All other steps including model building, training, fitting, and clustering are the same.

The Spark version of the application reads input images in a .jpg format from an HDFS directory. A single directory of images makes up an instance of an Image Clustering job. Spark's *binaryFiles* is used to create an RDD of all the images for a single Image Clustering job. The input images are read and converted to Spark

RDD of numpy arrays using the same transformations discussed earlier for the Flower Counter application. The numpy array is passed to a function which uses *cv2.xfeatures2d.SIFT* to extract features and computes descriptors from the input images. SIFT uses the Scale Invariant Feature Transform algorithm [39] to extract features from images. The extracted feature descriptors are filtered using Spark's *filter* transformation, and passed to the *K-Means* clustering function for model building, training and clustering. The clustering results are reduced into one RDD partition by using *coalesce*, and written to HDFS as a text file containing the names of the images and the cluster to which they belong. This application is CPU bound during K-means iteration stage and I/O bound during clustering stage.

3.6 Description of Experiments

The first experiment involves running a mixture of the sequential versions of the three image processing applications including Image Registration, Flower Counter and Image Clustering on *Onomi* and measuring the total makespan. This serves as the baseline value for the amount of time the applications require to run sequentially on a large server, and it is used for comparison with the distributed versions running on Spark. Since this experiment uses a single large server as a testbed, there is only one experimental setup, without any parameter tuning.

The next sets of experiments involves running a mixture of the distributed versions on the three cluster managers that support Spark, i.e. Standalone, Mesos and YARN, tuning various Spark configuration parameters and recording the performance metrics discussed earlier for analysis. Nonetheless, all the experiments only examined one factor at a time (except for the parameters that control Spark executor size), instead of multiple factors simultaneously. This means a particular parameter is selected for experimentation at a time while all the remaining configuration parameters are kept constant. Since Spark executor size is controlled by two parameters which consist of the number of CPU cores and the size of memory, these two parameters are configured together. This is a reasonable limitation because experimenting with all combinations of all factors would be extremely time consuming considering the amount of time it takes to run a set of the applications used in this thesis for a single experimental configuration. Furthermore, the experiments involve running the applications with and without speculative execution of tasks enabled, unless otherwise stated.

For all experimental setups, three instances of each workload applications are submitted at random, with five minutes delay between submissions. A total of nine jobs are submitted during each experimental run. The same seed number was used for the random number generation to allow having the same job submission sequence in each of the experimental runs. The sequence of jobs submissions is as follows: *imageClustering_job_1*, *flowerCounter_job_3*, *imageRegistration_job_2*, *imageClustering_job_2*, *imageRegistration_job_1*, *imageClustering_job_3*, *flowerCounter_job_2*, *flowerCounter_job_1*, and *imageRegistration_job_3*.

These sets of experiments would provide insights as to whether the image processing applications tested in this thesis are suitable for running on a distributed environment (such as Spark) and what performance

gains can be achieved by distributing the processing across multiple machines. In addition, the experiments would help in evaluating the performance of different cluster managers that support Spark, and the effects of the various Spark resource scheduling and allocation configuration parameters on applications performance.

3.6.1 Submitting Applications to Server and Cluster

A Python script is used for submitting the sequential version of the workload applications to *Onomi*. The script submits three instances of each of the three applications at random, with five minutes delay between submissions. The nine application instances were randomly ordered and submitted to *Onomi*. Both the start and the end time of the experiment is recorded. In addition, the script also records the timestamp of the job submissions.

Similarly, a Python script is used for submitting the distributed version of the applications to the cluster manager. The script randomly selects and submits a single job to the cluster through the master node, with five minutes delay between job submissions. During each experiment, the script submits a single job, then sleeps for five minutes before submitting the next job. This process repeats until all jobs are submitted. The nine application instances were randomly ordered, with the same seed in each run to ensure the same submission sequence. Also, the start and end time of the experiments, as well as the timestamps of job submissions are recorded for later analysis.

3.6.2 Experimentation Parameter Settings

This section describes the different parameter settings used for the experiments in this thesis, including parameters that control resource scheduling and allocations, various configurations for executor size and the number of parallel tasks per job, and parameters that control speculative execution of tasks. A full list of the various parameters tested and how their values were set is provided in Table 3.6. All the experimental sets were tested on the Standalone, Mesos and YARN cluster managers using both static and dynamic resource allocation.

Static Resource Allocation Experiments

The objective of these set of experiments is to analyze the performance of the default (that is static) scheduling and resource allocation behaviour of the three cluster managers that support Spark, and to compare the default Spark performance with the baseline results of running the sequential version of the workload applications on *Onomi*. In addition, the experiments investigate how the various Spark configuration parameters such as number of parallel tasks per job, resource size per executor (such as large number of executors with minimum resource per executor, and small number of executors but with large resource per executor) affect the performance of the three cluster managers tested when using the default static resource allocation.

These experiments are subdivided into the following categories according to resource configurations per executor and the size of Spark RDD partition (that is the number of parallel tasks per job):

Table 3.6: Parameter Configurations for all Experimental Sets

Experiment Set	Executor Resource Size	RDD partition size	Tasks Speculation
Set 1	1 CPU/2 GB	128 MB	True
Set 2	1 CPU/2 GB	64 MB	True
Set 3	1 CPU/2 GB	32 MB	True
Set 4	1 CPU/2 GB	128 MB	False
Set 5	1 CPU/2 GB	64 MB	False
Set 6	1 CPU/2 GB	32 MB	False
Set 7	3 CPU/6 GB	128 MB	False
Set 8	3 CPU/6 GB	64 MB	False
Set 9	3 CPU/6 GB	32 MB	False
Set 10	6 CPU/12 GB	128 MB	False
Set 11	6 CPU/12 GB	64 MB	False
Set 12	6 CPU/12 GB	32 MB	False

1. Set 1 CPU core and 2 GB memory per executors: The *spark.executor.cores* and *spark.executor.memory* are used to control the number of CPU cores and memory to use per Spark executor process respectively.
    - Set 128 MB per Spark RDD partition using the *spark.files.maxPartitionBytes* configuration parameter,
    - Set 64 MB per Spark RDD partition. This will double the number of tasks created by Spark when reading data from HDFS,
    - Set 32 MB per Spark RDD partition. This will quadruple the number of tasks created by Spark when reading data from HDFS.
2. Set 3 CPU cores and 6 GB memory per executors:
    - Set 128 MB per Spark RDD partition,
    - Set 64 MB per Spark RDD partition, and
    - Set 32 MB per Spark RDD partition.
3. Set 6 CPU cores and 12 GB memory per executors:
    - Set 128 MB per Spark RDD partition,
    - Set 64 MB per Spark RDD partition, and
    - Set 32 MB per Spark RDD partition.

Dynamic Resource Allocation Experiments

The objective here is to understand how different Spark cluster managers dynamically adjust resources based on applications requirements and the available workload when using the dynamic resource allocation feature of Spark, and to understand how this could affect resource utilization and application run time. Under these experiments, dynamic resource allocation was enabled on the three cluster managers. However, all other configuration parameters related to dynamic resource allocation were set to their default values. The experiments investigated how the number of tasks per job influences dynamic resource allocation behaviour of each Spark cluster manager, and analyze their performance and overhead in terms of cluster resource utilization and jobs completion times. The same set of applications and parameter setup as in the previous section were used, including resource configurations per executor, RDD partition size and speculative execution of tasks.

3.7 Summary

The chapter contains a detailed description of the hardware and software configurations of the two testbeds used in this thesis, including a large server and a heterogeneous Spark cluster, as well as the various Spark parameters tested, including parameters to control resource scheduling and allocation, number of parallel tasks per job, speculative execution of tasks and resource size per executor. Several metrics used for evaluating performance including makespan, CPU and Memory utilization, and job waiting and execution times were presented. Furthermore, a discussion of the three image processing applications tested was provided, as well as the characteristics of the datasets used for each of the applications instances. Finally, the multiple experiments done on the testbeds in order to evaluate the amenability of running image processing applications on Spark, and to quantify the performance gain of distributing the processing of the applications across multiple nodes were described in detail.

Chapter 4

Experimental Results and Discussion

This chapter contains the results of experiments described in Chapter 3. The first part of the chapter provides the analysis of results from a set of experiments done to characterize the resource requirements of each of the three parallelized versions of the image processing applications used in this thesis. The next part compares the makespan of a sequential version of the three benchmark image processing applications running on a server equipped with a large amount of CPU and memory resources, and the distributed version running on Spark without any parameter tuning (i.e. using all the default values provided by Spark). In addition, a number of experiments were established to evaluate the differences in makespan between multiple runs of the same experimental setup. Finally, the chapter investigates the impact of different Spark configuration parameters, including parameters to control job scheduling and resource allocation, Spark RDD partition size, speculative execution of tasks and the number of parallel tasks per job, and their effect on cluster resource utilization and applications makespan.

4.1 Workload Applications Resource Characteristics

A set of initial experiments were done to characterize the resource requirements of each of the distributed versions of the workload applications used in this thesis. The motivation for analyzing the resource requirements is to characterize the applications based on their resource utilization in a precise manner. This analysis is necessary in order to gain an initial insight into each application's resource usage and to understand the resource requirements of a mixture of multiple instances running concurrently.

The three distributed versions of the workload applications described in Section 3.5 were run in isolation on the 12-node Spark cluster described in Section 3.1. The two most important resources that Spark applications request from the cluster managers are CPU cores and memory. Hence, CPU and memory usage of each application across the entire cluster were monitored using the Ganglia cluster monitoring system when they are running individually, and used as performance metric for evaluation. The parameter settings used for the experiments in this section are listed below:

- Cluster Manager: Standalone,
- Resource Allocation Mode: Static,

- Resource per Spark executor: 1 CPU core, 2 GB memory,
- Spark RDD partition size: 32 MB,
- Speculative execution of tasks: Disabled.

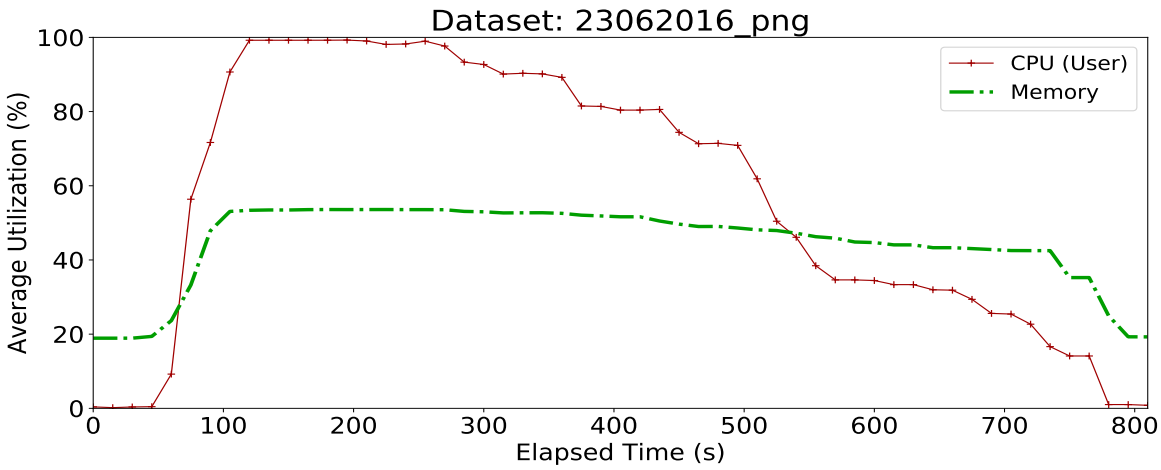
Spark Standalone cluster manager was chosen because when the resource allocation mode is set to static on this cluster manager, it allocates all the available resources to the submitted application. This will give an upper limit of what resources each application requires and how it utilizes the resources allocated when it is not sharing with other competing applications. The Spark RDD partition size was set 32 MB in order to have more tasks available.

Figure 4.1 shows the CPU and memory resource requirements of each of the three image processing applications used. The figures show that each of the three applications has a unique pattern of CPU and memory usage when running individually on the cluster.

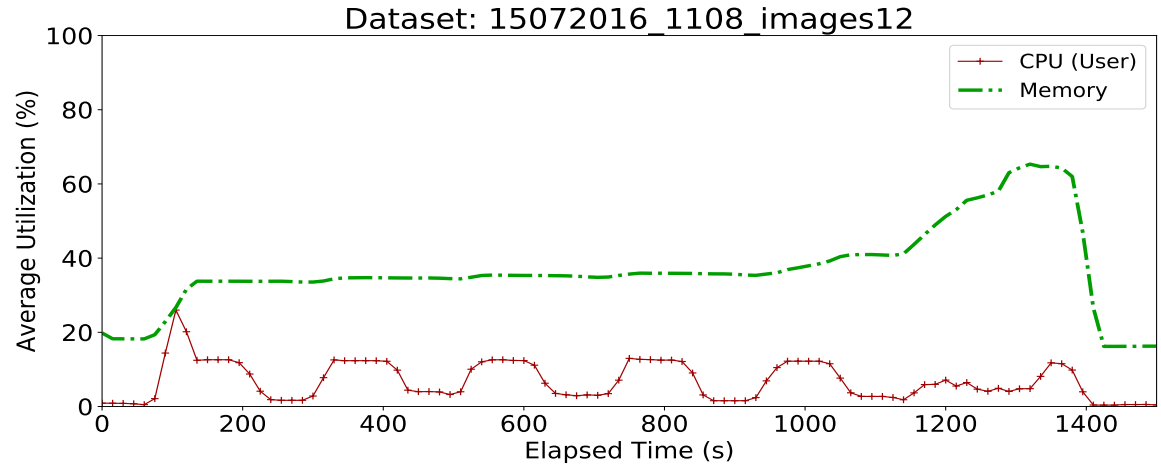
Figure 4.1a shows that the Image Registration application is highly CPU intensive. Right after the application was submitted, Ganglia reported the CPU usage on the cluster was close to 100%. This percentage starts to decrease gradually as the submitted tasks complete execution. The memory usage remains stable at around 50% until the number of tasks running decreases towards the end of the application. The high CPU requirement of this application is due to the stages in the application that compute keypoints and features in each of the images to register, and then use the keypoints to match images from different channels.

Figure 4.1b shows the CPU and memory usage of the Flower Counter application. The peaks in CPU usage observed in the figure were a result of Spark executing the stages that require CPU cycles for processing. The figure shows that during the stages that required CPU processing, the application's CPU usage never exceeds 30%. At multiple times during the execution, Ganglia reported the CPU metrics to be around 2%. On the other hand, memory usage stays at around 35% during the first five stages of the application. However, at the last stage when the application collects all processed data from each node and writes it to HDFS, the memory usage gradually increases to around 70%. This stage consists of many Spark operations such as *coalesce()*, which can shuffle data and return a new reduced Spark RDD with the user specified number of partitions, and *saveAsTextFile()*, which saves the content of the RDD to HDFS as a text file, using string representations of the elements. The output text file is very small in terms of size since it contains one line (consisting of the image name and the estimated number of flowers) per input image.

The Image Clustering application has different resource requirements than the other two applications discussed above. Figure 4.1c shows this application has two stages with high CPU requirements. The first peak in CPU usage metric was during building and training of the *K-Means* clustering model, and the second peak was during clustering of the images. Nevertheless, the CPU usage during these peak stages never exceeds 60%. There are also stages that Ganglia reported CPU usage to be around 10%, such as between training the model and classifying images, and when writing classification results to HDFS. On the other hand, the memory usage of this application gradually grows to about 50%, and stays between 50% to 60% during the



(a) Image registration resource usage



(b) Flower counter resource usage

(c) Image clustering resource usage

Figure 4.1: CPU and memory resource usage: Standalone/static/1 core/2 GB/32 MB/no speculation

first stage of the application. During the clustering stage, however, an increase in the memory usage to around 70% was observed. This slightly decreased when writing results to HDFS.

4.2 Performance of the Sequential and Distributed Applications

Recall that the initial goal of this thesis is to parallelize the three image processing applications discussed in Section 3.5 and determine if they are suitable for running simultaneously in a distributed environment such as Apache Spark. Therefore, the first set of experiments compare the makespan of the sequential and the distributed version of the applications.

The goal is to investigate the amount of resources that are required for the distributed version of the applications running on Spark compared to the sequential version running on a server equipped with large amount of resources as discussed in Section 3.1. As already discussed, the sequential versions of the applications were written in pure Python without any parallelism, while the distributed versions were written to run on Spark using *pyspark*.

The jobs submission scripts discussed in Section 3.6.1 were used to submit nine sequential and distributed versions of the workload applications. Three instances of each of the applications were submitted to *Onomi* and the Spark cluster managers (that is Standalone, Mesos, and YARN), with 5 minutes inter-arrival between job submissions, and using the same submission sequence. Only a single run of each experimental setup was done and used for the analysis in this section. Recall that *Onomi* has 56 virtual cores and 620 GB memory, while the testbed cluster is equipped with 77 virtual cores and 185 GB memory. In addition, the experiments on Spark used the following configuration settings:

- Cluster Manager: Standalone, Mesos, YARN,
- Resource Allocation Mode: Static,
- Resource per Spark executor: 1 CPU core, 2 GB memory,
- Spark RDD partition size: 128 MB,
- Speculative execution of tasks: Disabled.

With these configurations, Standalone, Mesos and YARN used 66 Spark executors, each equipped with 1 core and 2 GB memory. While *Onomi* used 56 cores and 620 GB memory. It is important to note that although 185 GB memory is available on the cluster, not all the memory is utilized by Spark applications. This is because the number of CPU cores on the node as well as the values specified for *spark.executor.cores* and *spark.executor.memory* determined the maximum amount of memory that could be used by applications. For example if a node is configured to provide Spark with 7 CPU cores and 34 GB memory, leaving 1 CPU core and 1 GB memory for the Operating System (such as the configuration on two of the nodes in the testbed cluster used), and each Spark executor is set up with the resource size of 1 CPU core and 2 GB memory (such

as the configuration for the experiments in this section) using *spark.executor.cores* and *spark.executor.memory* respectively, the maximum amount of memory that could be used by applications scheduled on this node is 7*2=14 GB. This is because for every 1 CPU core, only 2 GB would be used. Since memory can not be shared between nodes of the cluster, the remaining 20 GB memory would not be used. The same concept applies to all Spark experiments in this thesis. Thus, for all practical purposes, the highest amount of memory Spark could use on any of the cluster nodes is 14 GB.

Figure 4.2 shows the makespan value of the nine jobs. The figure shows that the distributed version of the applications did worse on the Standalone cluster manager when using static resource allocation, which had the effect of preventing any simultaneous operation for the Standalone. Although the cluster has more CPU cores than *Onomi*, the distributed version of the applications took 20636 seconds to complete on Standalone, approximately 21% more than what the sequential version took on *Onomi*. This is expected because when Spark uses the Standalone cluster manager with default resource allocation mode, multiple applications submitted are run sequentially in FIFO order. In addition to the default (static) resource allocation mode, the values set for other configuration parameters contributes to the high makespan observed on the Standalone. In particular, the 128 MB RDD partition size limits the number of tasks that could run concurrently during the execution of each of the applications. Although Spark executes the tasks within the applications in parallel, the number of tasks that can execute during any stage are limited by the amount of data to process and the number of bytes to pack per RDD partition.

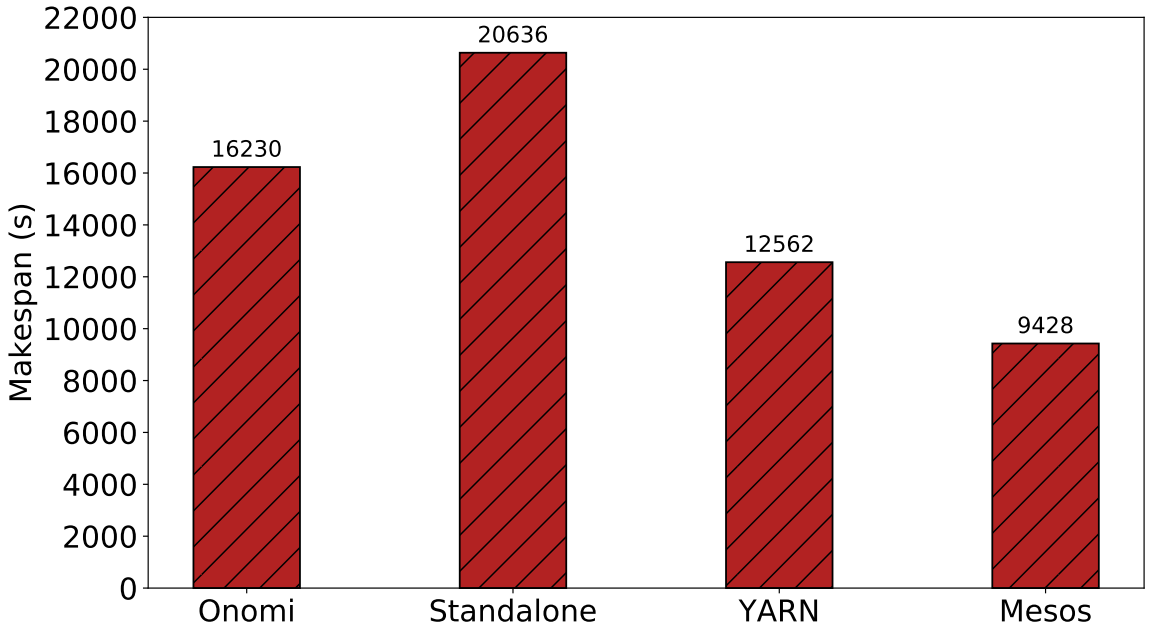


Figure 4.2: Initial Makespan Measurements - Cluster and Single Machine

With the FIFO scheduling mode on the Standalone cluster manager, the execution time of all subsequent applications includes the waiting and running time of all previously submitted applications. For these sets of experiments on Spark with default job scheduling and resource allocation mode, using the 128 MB RDD partition size results in fewer tasks in each stage compared to when using a smaller RDD partition size. In particular, the *flowerCounter_job_1* has 314 MB of data to process. With 128 MB RDD partition size, Spark created 3 tasks in each stage of this application. This means that even though the application was allocated 77 executors, each equipped with 1 CPU core and 2 GB memory (that is all the cluster resources according to the default behaviour of the Standalone cluster manager), it can only use 3 out of the 77 allocated executors at each stage throughout its execution. Therefore, approximately 96% of the cluster resources are completely idle during the execution of this application, meanwhile other application that were submitted during this time are queued and waiting for resources. This is obviously not a wise way to schedule jobs and resources.

Although the multiple applications submitted to *Onomi* are sequential, they each run as separate Python processes and are scheduled using the operating system's default process scheduler, which aims to maximize overall CPU utilization as well as performance. Applications do not need to wait for resources to be released from other applications in order to run. This explained why *Onomi* shows better performance than Spark with the Standalone cluster manager in Figure 4.2. In addition, at multiple stages during the execution of the sequential applications, Python spawn multiple threads which use a large percentage of the CPU resources of *Onomi*. For example, when a single instance of the Image Registration application was run in isolation on *Onomi*, it used up to 99% of the 56 cores available on the machine during calls to OpenCV functions.

On the YARN cluster manager, however, the distributed version of the applications took 12562 seconds to complete, which is approximately 23% less time when compared with *Onomi*. This is expected as well, because YARN by default uses the *CapacityScheduler* to manage resources. Recall that this scheduler is designed to manage resources as a shared and multi-tenant cluster, while maximizing the throughput and the cluster utilization. Although when the Spark resource allocation mode is set to static, the YARN cluster manger would allocate only three containers to each submitted application, that is one container to run the *ApplicationMaster* and the other two for executing tasks. Nevertheless, applications do not need to wait for a long time to receive resources compared to using FIFO on Standalone.

The Mesos cluster manager has the best performance compared to *Onomi*, Standalone, and YARN. The distributed version of the applications took 9428 seconds to complete on Mesos, which provides approximately 42% reduction in makespan compared to *Onomi*. This is because Mesos by default uses the Dominant Resource Fairness algorithm to fairly share cluster resources among applications. Nevertheless, the first job submitted was allocated all the cluster resources. Recall that this is the default behaviour on Mesos when using static resource allocation mode. During the time when the first submitted job was running, the cluster resource utilization was very low, and all the subsequent jobs submitted were queued, since Mesos can not remove resources from an actively running Spark application when using static resource allocation. However when the first job completes, Mesos fairly shares all the available resources among the applications waiting

for resources. This substantially reduces the time applications spend waiting for resources, which in turn improves makespan and overall cluster resource utilization. From the analysis of results in this section, the following insights were obtained:

- The times individual jobs spend waiting for resources greatly impacts the total makespan of all submitted jobs. Therefore, it is very important to minimize the jobs' waiting time in order to improve total makespan.
- Although *Onomi* shows better performance than the Standalone cluster manager with the default parameters, the distributed version of the workload applications submitted to Mesos and YARN (using the default scheduling and resource allocation mode) show great performance improvements when compared with the sequential version.

The default values for the various resource scheduling and allocation configuration parameters of Spark have been shown to provide poor performance when multiple applications need to run concurrently and share cluster resources efficiently. For example, the Spark documentation[1] encourages setting the *spark.cores.max* or *spark.deploy.defaultCores* configuration parameters to a low value on a shared cluster to prevent applications from monopolizing the entire cluster by default. In addition, the experiments in this section show that simply using a different cluster manager to schedule and allocate resources to Spark application can substantially improve performance. Therefore, these preliminary experiments provide the motivation for running further experiments with different values for the chosen Spark configuration parameters in order to understand how jobs makespan performance and overall resource utilization of the distributed version of the workload applications can be improved.

4.3 Evaluating Makespan Differences of Multiple Runs

During the course of running the experiments, there were several times when the testbed cluster running Spark, Mesos and YARN needed to be restarted due to various reasons. In addition, there were concerns about how the makespan changes between multiple runs of the same experimental setup. To investigate how running experiments on a freshly started cluster and a cluster that has been running for quite some time and possibly cached data and Spark libraries, and also to quantify the difference in makespan between multiple runs of the same experimental setup, a set of experiments was done using the same parameter setup discussed in Section 4.1, except the experiments in this section were run on all the three cluster managers.

In these experiments, Standalone, Mesos and YARN cluster managers were first restarted. Three instances of each of the Image Registration, Flower counter and Image Clustering applications were submitted at random using a simple script that employed a random number generating function to emulate random submission of applications to the desired cluster manager. An inter-arrival time of 5 minutes was set between

[1] `https://spark.apache.org/docs/2.1.0/spark-standalone.html` (03 Jan, 2018)

job submissions. In addition, the same seed was used for generating the random number, to allow having the same submission sequence for each run. Five runs were done sequentially on each of the three cluster managers using the same configuration settings. For example, the first run on the Standalone was done on a freshly started cluster, followed by four subsequent runs. The same concept applies to YARN and Mesos.

Figure 4.3 shows the makespan results of multiple runs of the same experimental setup on the three cluster managers. The figure shows the first run deviated from the rest of the four subsequent runs in all the configurations. The percentage difference between the first and the second run is approximately 11% on Standalone, 14% on YARN and 3% on Mesos. On the other hand, the difference between the second and the subsequent runs is less than 1% in all cases except for the fourth and the fifth run on YARN cluster manager, which is approximately 2%.

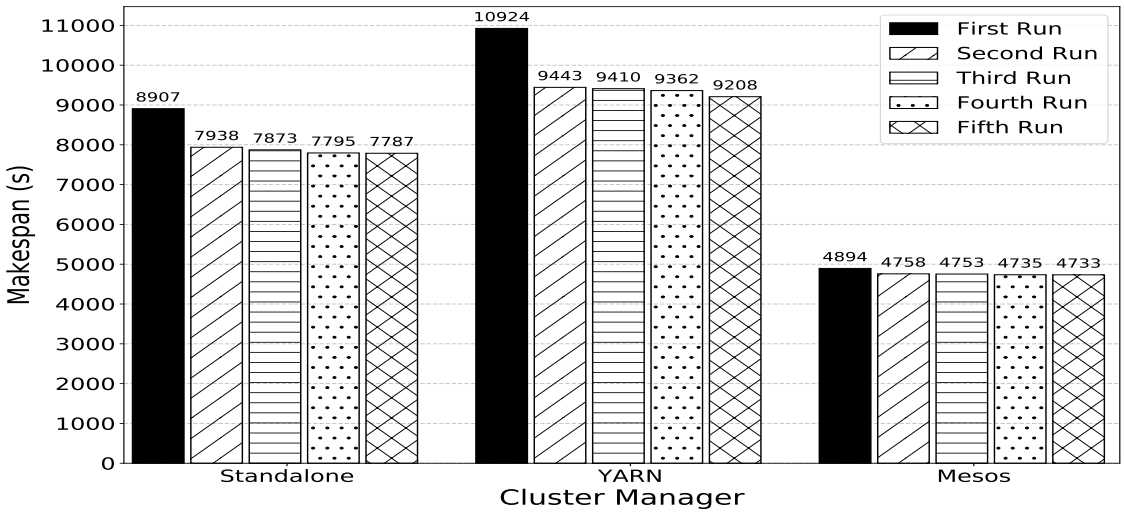


Figure 4.3: Multiple runs of the same experimental setup: Static/1 core/2 GB/32 MB/no speculation

A *t-test* with 3 degrees of freedom and α value of 0.05 was used to show that the first run deviates significantly from the rest. The mean and standard deviation of the subsequent (that is second to fifth) runs was calculated and used in the *t-test*. The Standalone, YARN and Mesos has *t-scores* of 29.69, 30.17, and 23.68 respectively. Comparing the scores with the *t-value* of 2.353 shows that the score of the first experimental run on all the three cluster managers is significantly greater than the *t-value*. Therefore, the first run varies significantly from the rest of the experimental runs.

Due to this high variation between the first run on a freshly started cluster and the subsequent runs, whenever any of the three cluster managers was restarted, an initial run was done to warm up the cluster. The results of this initial first run are not included in any of the analysis. This concept of warming up the cluster by doing an initial run agrees with Venkataraman *et al.* [58] who performs a warm up run in order to trigger the JVM's just-in-time compilation.

The three image processing applications used in this thesis required a very long time to run. A substantial amount of time is needed in order to do multiple runs of each of the many different experimental configurations described in Chapter 3. Since the variation in makespan between the second and subsequent runs is experimentally shown in Figure 4.3 to be relatively small, only one run for each experimental setup was done and used for the analysis.

4.4 Impact of Spark Resource Allocation Modes on Makespan

A number of experiments were conducted to understand the effects of the two resource allocation modes supported by Spark: static and dynamic. For all the experiments reported here, each Spark executor was set to have 1 CPU core and 2 GB memory. Three different instances of Image Registration, Flower counter and Image Clustering applications were submitted in random order, with 5 minutes delay between job submissions. Three different values of Spark RDD partition sizes were tested including 128 MB, 64 MB, and 32 MB. The cluster CPU and memory utilization, and jobs makespan were used as performance metrics. The Spark parameters varied and the different values tested in the experiments are listed in Table 4.1. The descriptions of these parameters were provided in Table 2.1 and Table 2.3:

Table 4.1: Parameters Varied and the Values Tested

Configuration parameter	Values Tested
spark.files.maxPartitionBytes	128 MB, 64 MB, 32 MB
spark.executor.memory	2 GB
spark.executor.cores	1
spark.speculation	False
spark.dynamicAllocation.enabled	False, True

For the resource utilization graphs, only the results of 128 MB RDD partition size, without speculative execution of tasks is presented here. This is because each of the different experimental setups on the same cluster manager shows similar CPU and memory utilization pattern when using the same resource allocation mode, regardless of the other parameters tested.

4.4.1 Resource Utilization Under Static Resource Allocation

Standalone, Mesos, and YARN use static resource allocation by default to allocate resources to submitted applications. Although the resource allocation mode is the same, experimental results show that each of the three Spark cluster managers has different resource utilization patterns when using static resource allocation. This is because they each implement different resource scheduling and allocation policies. Therefore, the motivation of this experiment is to understand the default behaviour of each of the cluster managers considered,

and to explain empirically how the difference can affect applications performance.

Spark Standalone cluster manager uses FIFO to schedule the submitted applications, and each application was allocated the entire cluster resources. Figure 4.4 shows the CPU and memory utilization of the standalone cluster manager. As shown in the figure, the submitted applications run in sequence. During the execution of the Image Clustering application instances, the CPU utilization of the cluster is around 15%, while memory utilization gradually increases to approximately 50%. Similarly, each of the Flower Counter application instances shows very low CPU utilization of less than 10%, and a maximum memory utilization of approximately 52%. The three peaks in CPU utilization are during the execution of Image Registration, which is the most CPU intensive of three image processing applications used. The figure shows that the CPU utilization on the cluster suddenly increased to well over 50% when executing the instances of the Image Registration application. Similarly, memory utilization of this application was approximately 42% in all the instances.

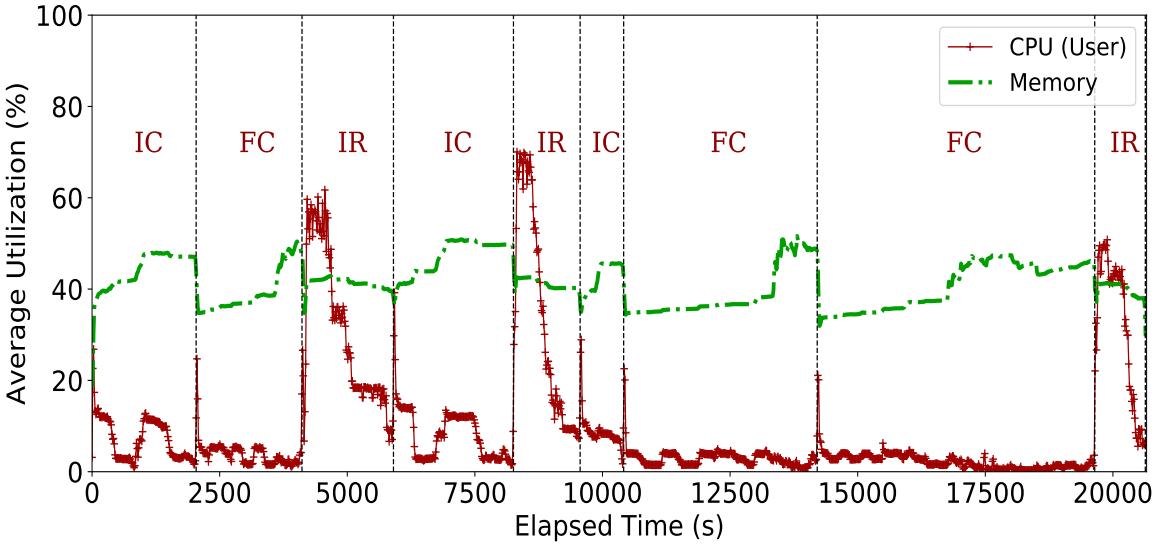


Figure 4.4: Resource utilization of Standalone/static/1 core and 2 GB per executor/128 MB RDD partition size/no speculation

Overall, using the default static resource allocation with the Standalone cluster manager underutilizes the cluster and results in unnecessary increases in application execution time. This is clearly not efficient in a multi-user cluster because the total execution time of each subsequent application will include the sum of the execution times of all previous applications (i.e. the times they spend waiting for resources to become available) plus their actual execution times (i.e. from the time they receive the first resources for execution), therefore, increasing the total makespan from submission of the first job until the last job is complete.

The CPU and memory utilization of YARN cluster manager using static resource allocation is shown in Figure 4.5. The figure shows that both CPU and memory utilization gradually increase to approximately 40% as more applications were submitted. After about 6000 seconds have elapsed, only one application

(*imageClustering_job_2*) is left running. At this time, the CPU utilization of the cluster is less than 5%, while memory utilization is around 20%. In such a situation, a new batch of jobs could be accepted and use the basically idle cluster.

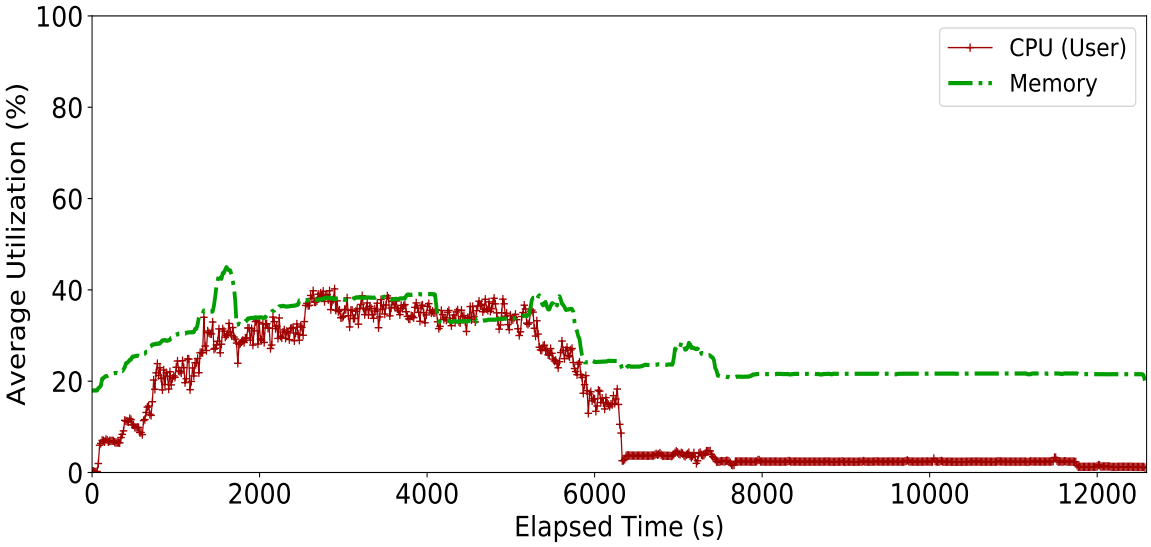


Figure 4.5: Resource utilization of YARN/static/1 core and 2 GB per executor/128 MB RDD partition size/no speculation

Recall that YARN allocates 3 containers (1 for executing the *ApplicationMaster* and the other 2 for executing tasks) to each submitted application by default. The cluster used in this thesis is set up to have 77 executors with 1 CPU core and 2 GB memory for this experiment. During the experiment, a total of 9 applications were submitted. Therefore, the YARN cluster manager has enough executors to allocate to all applications. Although the default scheduling behaviour of YARN using static resource allocation executes applications in parallel and reduces the waiting time of individual applications, it substantially underutilizes cluster resources. Since all the applications used for this experiment have more than 2 tasks, Spark uses a fair scheduler within each application's *SparkContext* to schedule and run tasks concurrently on the allocated executors. However, the number of tasks that can run concurrently are limited to the number of executors, which in this case is 2. This increases both the execution time of every individual application and the total makespan. In order to increase the number of executors that YARN could allocate to Spark applications using the static resource allocation, the cluster administrator must manually set the *spark.executor.instances* configuration parameter to the appropriate value. Manually setting this parameter is not easy to do, however, since the optimal number of executors that an application needs depends on the application type and the size of the data to process. Therefore, setting a high value for this parameter will introduce similar problems observed in Standalone, including underutilizing the cluster resources and increasing makespan. This is because YARN may allocate more executors to applications than they actually need. In addition, other applications may have to spend a long time waiting for resources to be released since YARN can not reclaim

resources from an actively running Spark application when using static resource allocation mode.

Figure 4.6 shows the resource utilization of Mesos cluster manager using static resource allocation. With this scheduling mode, Mesos allocates all the cluster resources to the first application submitted. The application holds these resources throughout its execution. As a result, the CPU utilization of the cluster is below 20%, while memory is approximately around 40% during the execution of the first application. All subsequent applications submitted during this time were queued. When the first application finished execution, Mesos reclaims the allocated resources and divides them among all the queued applications waiting for resources. This immediately increases the CPU utilization to about 80% and memory to approximately 60%. Whenever there are resources available, Mesos uses the Dominant Resource Fairness algorithm to fairly share the resources among all waiting applications. Nevertheless, when only one application is waiting, it gets all the resources regardless of its resources requirements. Therefore, when the Mesos cluster manager uses static resource allocation, it can only share resource among multiple applications if there is more than one application waiting for resource allocation at the time resources become available.

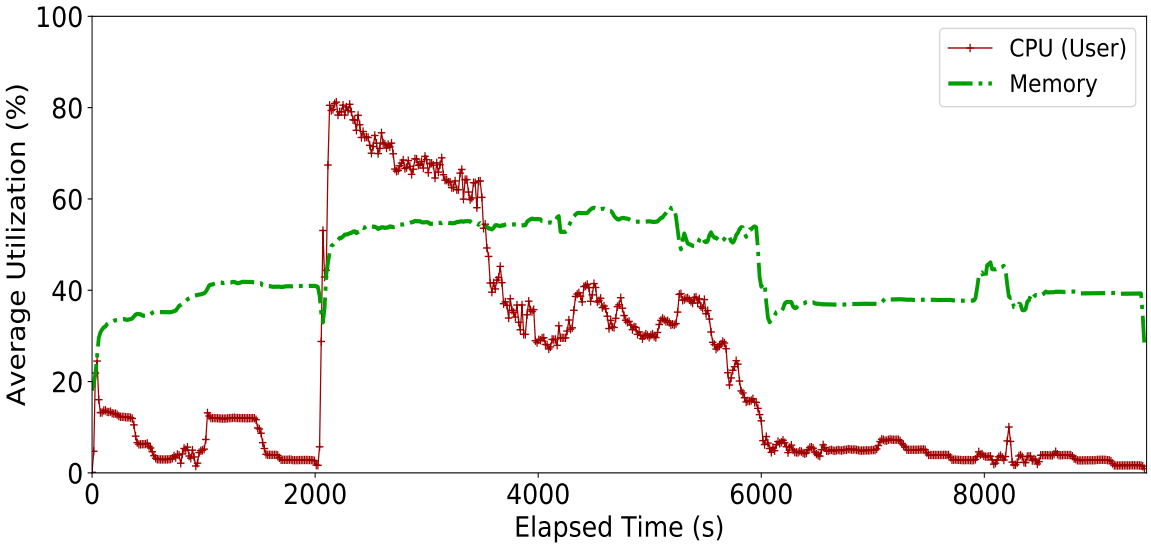


Figure 4.6: Resource utilization of Mesos/static/1 core and 2 GB per executor/128 MB RDD partition size/no speculation

The results discussed above shows that static resource allocation is not efficient for sharing cluster resources among multiple applications on all the three cluster managers tested. This mode under-utilizes the cluster resources, and unnecessarily increases the makespan of the applications.

4.4.2 Resource Utilization Under Dynamic Resource Allocation

Spark provides the dynamic resource allocation mode in order to address the issues accompanying the static resource allocation. The same sets of experiments discussed for the static resource allocation were done using

dynamic resource allocation. All other configuration parameters related to dynamic resource allocation were set to their default values. The results for resource utilization are discussed here, while makespan analyses is presented later in Section 4.4.4.

Figure 4.7, 4.8 and 4.9 present the CPU and memory utilization of the Standalone, YARN and Mesos cluster managers respectively. The three graphs shows there are similar pattern of resource utilization on all the three cluster managers when using this mode. This is because they all use the same resource allocation policy to dynamically adjust the resources applications occupied based on the current workload (i.e. the number of the tasks at a particular stage). When the cluster manager allocated more executors than needed by the application or when some tasks are done executing, the application gave back the unused resources to the cluster manager.

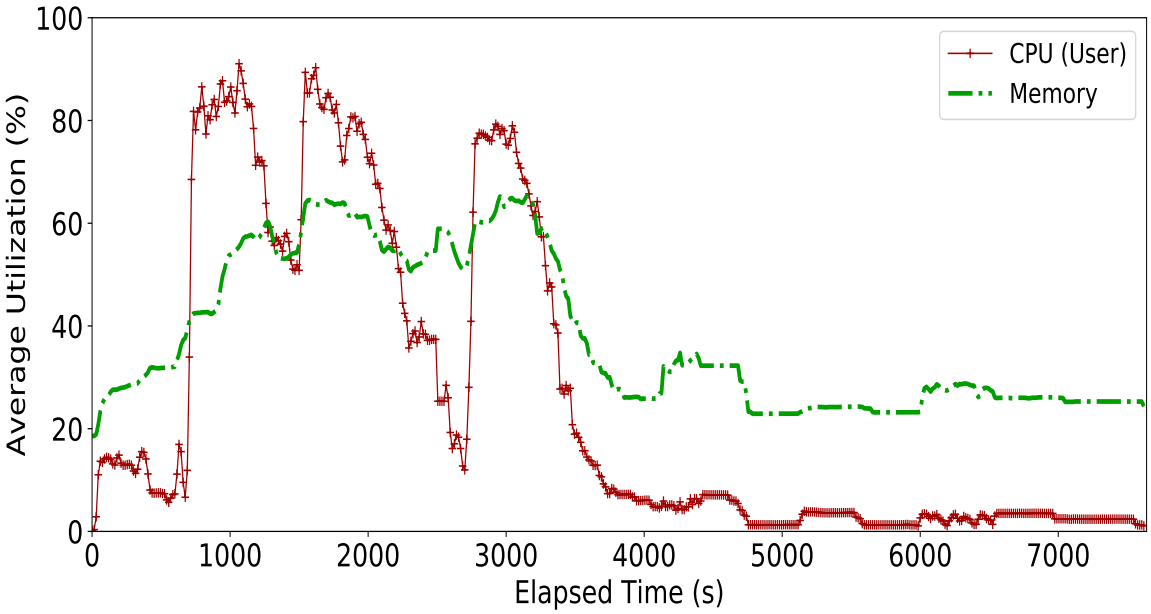


Figure 4.7: Resource utilization of Standalone/dynamic/1 core and 2 GB per executor/128 MB RDD partition size/no speculation

With the dynamic resource allocation feature enabled, both the CPU and memory utilization fluctuate on all the three cluster managers tested throughout the execution of the experiments. During the highly resource intensive stages such as when executing the instances of the Image Registration application, the cluster resource utilization rise to approximately 85% for CPU and 60% for memory. These stages include between approximately t=900 seconds to t=2100 seconds, and also between approximately t=2900 seconds to 3600 seconds. The values reduce to below 40% for both CPU and memory during the execution of low resource intensive stages. This shows the dynamic resource allocation is capable of scaling resource utilization up or down depending on the number of currently running tasks.

After t=4000 seconds, most of the resource intensive applications stages have finished executing and there

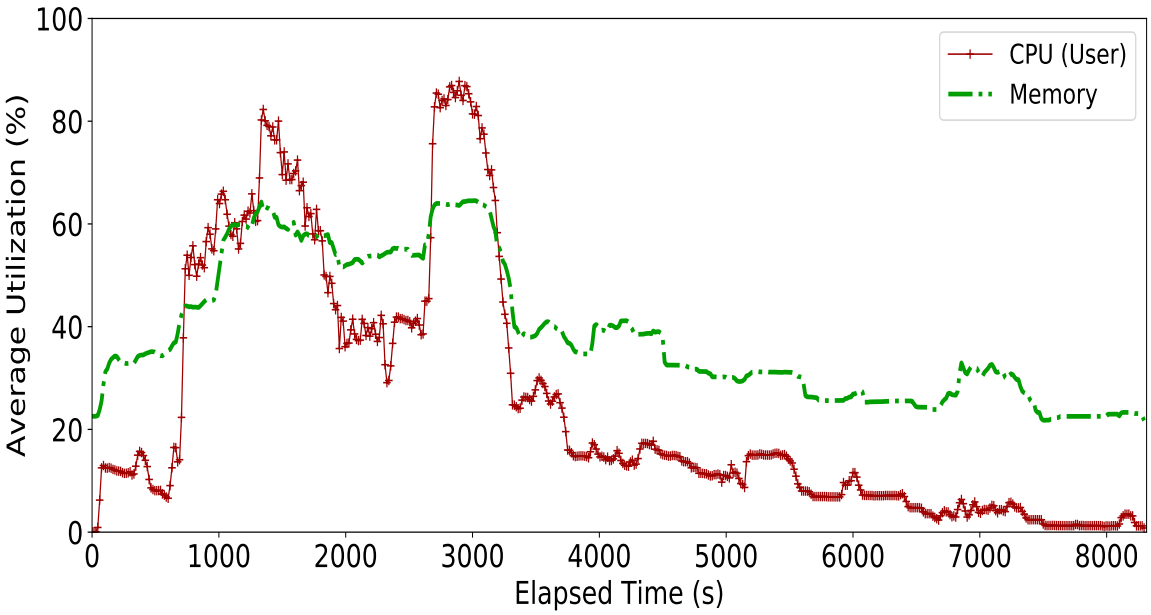


Figure 4.8: Resource utilization of YARN/dynamic/1 core and 2 GB per executor/128 MB RDD partition size/no speculation

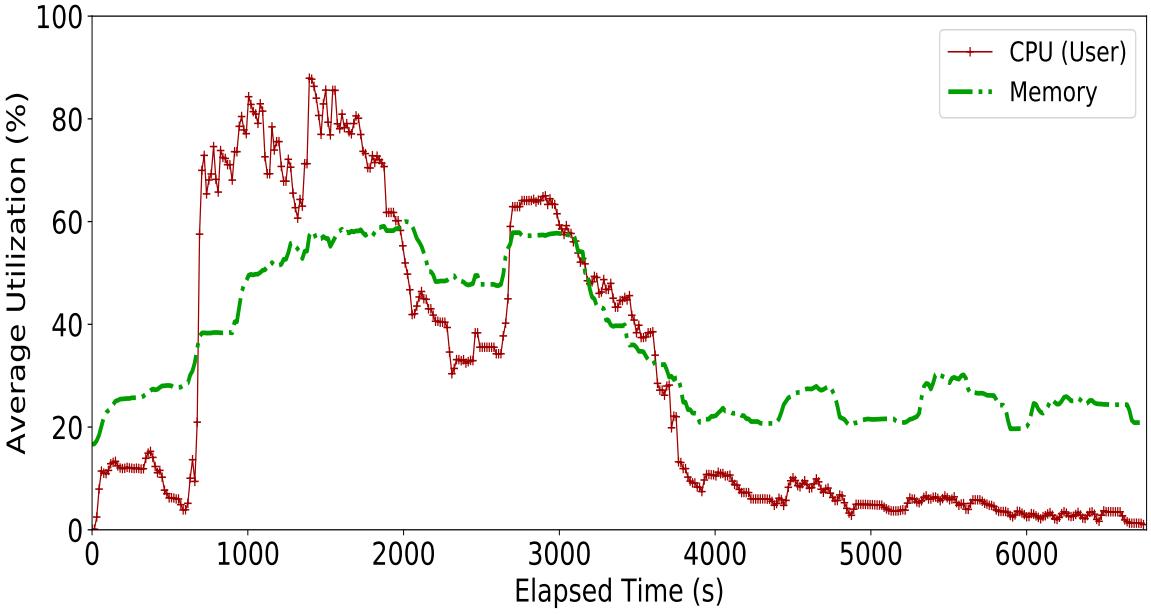


Figure 4.9: Resource utilization of Mesos/dynamic/1 core and 2 GB per executor/128 MB RDD partition size/no speculation

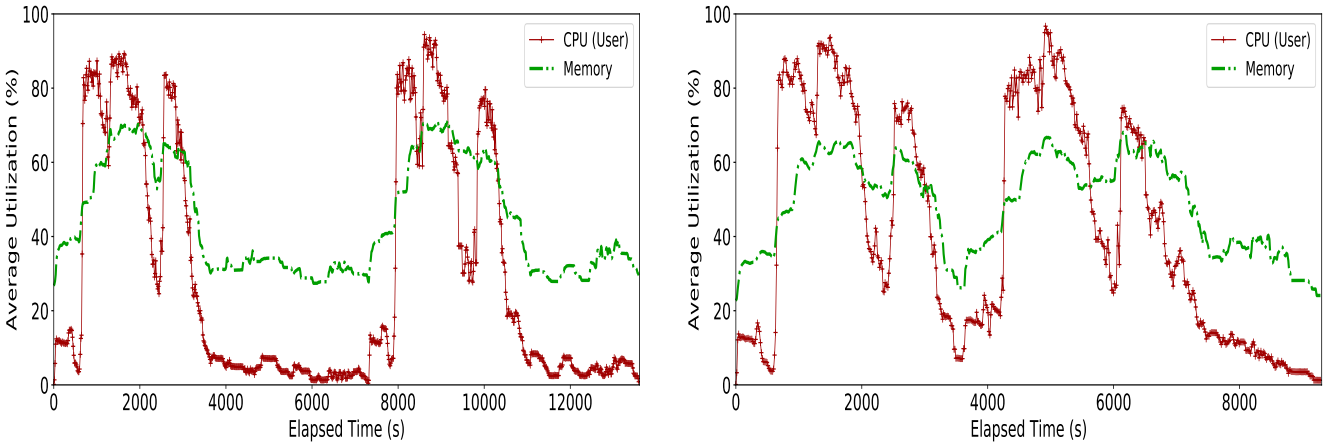
are no other applications waiting for resources. At this time, the CPU utilization suddenly drops to less than 10% on Standalone, and less than 20% on Mesos and YARN. Similarly, the memory utilization decreases to less than 40% on all the cluster managers. As a result of low resource utilization after t=4000 seconds, a new set of applications would need to be submitted to keep the cluster busy.

4.4.3 Submitting New Set of Jobs After Cluster Utilization Drops

The insights from the previous subsection provided the motivation to further investigate how the workload submission pattern affects cluster utilization when using dynamic resource allocation to schedule applications. Recall from the previous section that after about t=4000 seconds of submitting the workload applications to the Standalone cluster manager, approximately 90% of the cluster CPU is idle, and approximately 60% of the cluster memory is free.

Therefore, a set of experiments was done to evaluate the impact of workload submission pattern on makespan. In these experiments, there are two groups of workloads, each having 9 applications to process. The applications are 3 instances each of the image registration, flower counter and image clustering. A script was written that submitted the workloads to the Standalone cluster manager using two different patterns. The first pattern submitted the first workload group (that is 9 applications) and waited until all the applications in that group has finished executing before submitting the next set of applications. The second pattern submitted the first group of the applications and waited for 3600 seconds (a time chosen based on the observation of cluster utilization). After this time, the script started submitting the next set of applications.

Figure 4.10 shows the CPU and memory utilization of the Standalone cluster manager (using dynamic resource allocation with 128 MB RDD partition size) for the two sets of workloads. For the experiment where the second set of applications were not submitted until after all the applications from the first set has finished executing, the makespan was 13587 seconds. As shown in figure 4.10a, between t=3600s to t=7300s, the aggregate CPU utilization on the cluster is consistently below 20%, while memory utilization is below 40%. This shows there is very low resource utilization on the cluster during this period of time since only few applications are still running. At about t=7300s, the second set of applications were submitted. This is the reason for the sharp raise in CPU utilization from less than 5% to approximately 90%, and memory utilization from approximately 30% to approximately 70%. Figure 4.10b presents the cluster utilization of the experiment where the second workload was submitted exactly 3600s after submitting the first. For this experiment, Spark took only 9290 seconds to complete processing both workloads, reducing the makespan by approximately 32%. This shows that a substantial performance improvement may be achieved by submitting new set of applications after the utilization on the cluster drops below a certain threshold. However, the cluster needs to be monitored to determine the appropriate time for submitting new set of applications in order to avoid interference between workloads which can result in severe resource contentions and performance degradation. Providing such a system for monitoring is put in future work.



(a) Second workload set was submitted after the first set has finished executing

(b) Second workload set was submitted exactly after 3600s of submitting the first set

Figure 4.10: Resource utilization of two group of workloads: Standalone/dynamic/1 core and 2 GB per executor/128 MB RDD partition size/no speculation

4.4.4 Factors Affecting the Makespan

Makespan is a very important metric used to understand how different parameter settings affect system performance. The figures in this section compare the makespan for each of the experiments discussed earlier.

Makespan Analysis of Static Resource Allocation:

Figure 4.11 compares the makespan of various experimental runs under different sizes of Spark RDD partition when using static resource allocation without speculative execution of tasks. The figure shows the makespan value generally decreases when the size of Spark RDD partitions was decreased due to increase in the number of concurrently running tasks. The Standalone cluster manager provides the worst makespan results for the configurations with 128 MB and 64 MB RDD partition sizes, since there are relatively few tasks running concurrently. Therefore, this increases the runtime of individual applications and consequently the waiting time of subsequent applications. Although with the default configurations for the static resource allocation, YARN allocated only three containers per application (one container for executing the *ApplicationMaster* and the other two for executing tasks), it took approximately 61% and 75% of the makespan values observed on the Standalone when the RDD partition size was set to 128 MB and 64 MB respectively. Similarly, Mesos took approximately 46% and 38% of the makespan time of Standalone to finish executing all the applications on the configurations with 128 MB and 64 MB RDD partition size respectively. The reason is because applications spend much more time waiting for resources on the Standalone cluster manager than they did on YARN and Mesos. For example, with 128 MB RDD partition size, the average waiting time of applications submitted to Standalone is 7039 seconds, while the average execution time is approximately 2288 seconds. These values are 31 seconds and 4901 seconds on YARN. For Mesos, the average waiting time

is approximately 1101 seconds while the average execution time is approximately 3444 seconds. Similarly, when the Spark RDD partition size is set to 64 MB, the average waiting time on Standalone, YARN and Mesos is 6513 seconds, 31 seconds and 477 seconds, while the average execution time is 1779 seconds, 5783 seconds and 2025 seconds, respectively. This shows that even though individual applications take less time to execute on Standalone using static resource allocation with 128 MB and 64 MB RDD partition size, the waiting time substantially contributed to the high makespan value.

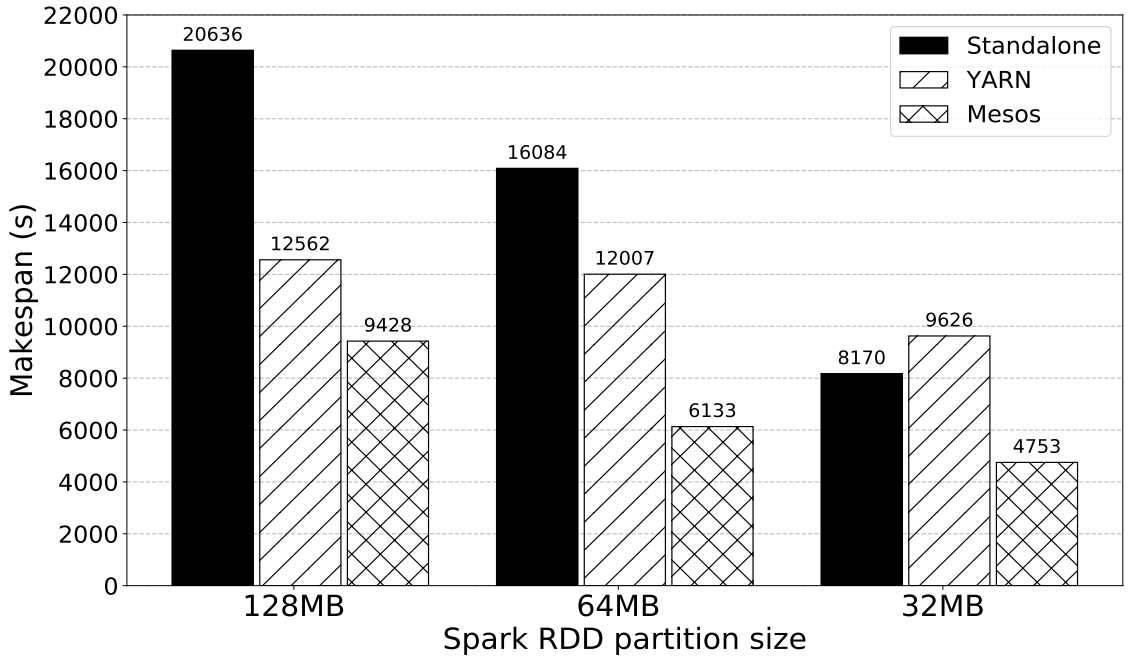


Figure 4.11: Makespan comparison: static/1 core/2 GB/no speculation

When the RDD partition size gets smaller, the number of tasks to schedule increases, therefore having more resources to execute more tasks concurrently reduces both the average waiting time and execution time of applications on the Standalone cluster manager. The results for the configuration with 32 MB partition size show that the Standalone cluster manager performs better than YARN. The reason is that there are only three executors per application on YARN regardless of the number of tasks. Therefore, doubling the number concurrent tasks (i.e. by changing the partition size from 64 MB to 32 MB) only reduces the makespan of YARN by 20%, while this value is 49% on Standalone.

The results for Mesos show that increasing the number of tasks by changing RDD partition size from 128 MB to 64 MB, and 64 MB to 32 MB has a large impact on the makespan, which was decreased by 35% and 23% respectively. Nevertheless, on all the configurations tested for static resource allocation experiments, the Mesos cluster manager produces the best results. This is because, even though static resource allocation is specified within Spark configuration settings, Mesos implemented some level of fair sharing of resources

internally. The policy implemented by default on Mesos implies that whenever resources becomes available, they should be fairly shared using Dominant Resource Fairness algorithm among all the applications that are currently waiting for resources. When there is only one application requesting resources, it will be allocated all the resources that become available.

It should be noted that when using static resource allocation, none of the cluster managers tested are able to remove the resources that were already allocated to applications during runtime. The application can only release the resources after it has finished executing all its tasks.

Makespan Analysis of Dynamic Resource Allocation:

Figure 4.12 shows the makespan comparison of various experimental runs under different sizes of Spark RDD partition when using dynamic resource allocation without speculative execution of tasks enabled. Comparing Figure 4.11 and 4.12, a similar pattern of reduction in makespan as the size of Spark RDD partitions decreases could be seen. This is expected because less data per RDD partition means more tasks can run concurrently.

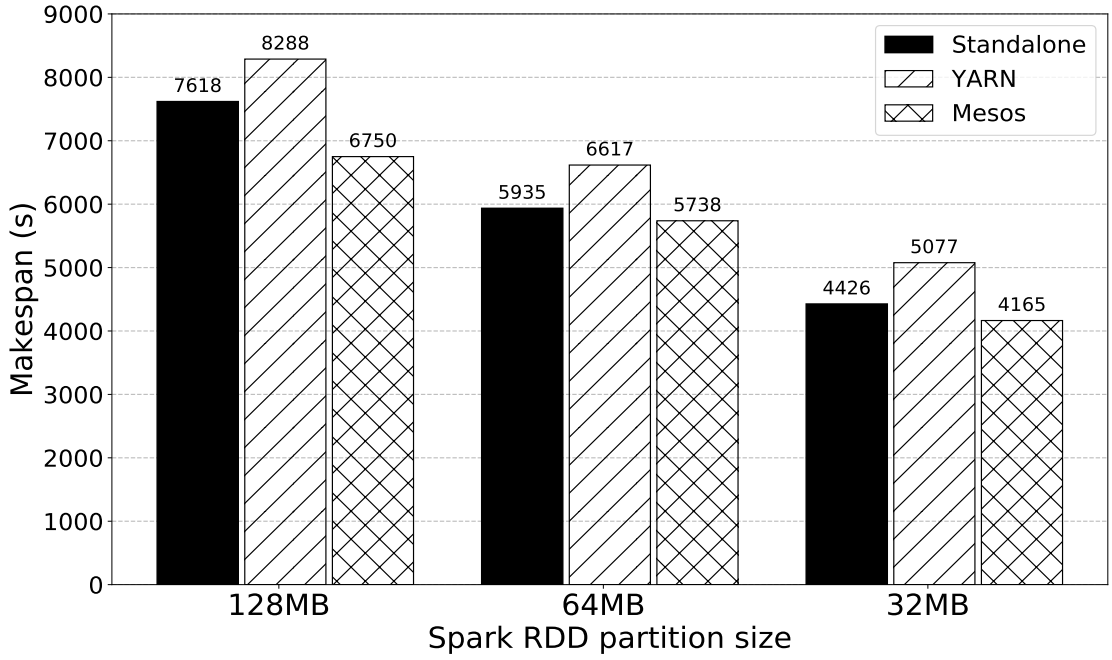


Figure 4.12: Makespan comparison: dynamic/1 core/2 GB/no speculation

Furthermore, when the makespan results of static resource allocation shown in Figure 4.11 are compared with the makespan results of dynamic resource allocation shown in Figure 4.12, the effect of dynamic resource allocation on makespan can be clearly seen. Figure 4.12 shows a substantial reduction in makespan values for all the experimental setups. On the configuration with 128 MB RDD partition size, the makespan of the jobs was reduced by approximately 63% on Standalone, 34% on YARN and 28% on Mesos when compared

to the values of static resource allocation. This is because with dynamic resource allocation enabled, the first application submitted to all the cluster managers was allocated only the exact number of executors it needed according to its number of tasks, therefore, reducing the waiting time of subsequent applications, and hence a reduction in the total makespan. For example, the average waiting time of the applications submitted to Standalone, YARN and Mesos are 17 seconds, 62 seconds and 49 seconds respectively on the configuration with 128 MB RDD partition size. The same reason applies for the results observed for 64 MB and 32 MB partition.

At any stage during the application's execution, whenever the number of tasks decreases, Spark requests the cluster manager to remove any executor that has been idle for more than 60 seconds and did not cache any data that could be needed by other executors. Similarly, when a stage has more tasks than the current number of executors allocated to it, and has pending tasks backlogged for more than 1 second, Spark requested more executors from the cluster manager. At this stage, if there are any available executors, the cluster manager allocated them to the requesting application. This is a concept used by all the three Spark supported cluster managers tested when dynamic resource allocation mode is enabled.

Another important thing to notice from the figure is that in all cases, YARN performs worse than the other two cluster managers. This is because for any application submitted to YARN, one container is first allocated to the application to run the *ApplicationMaster*. As already discussed in Section 2.2.3, the *ApplicationMaster* daemon is responsible for negotiating resources from the *ResourceManager* and working with the *NodeManager* to execute and monitor the tasks. This means that applications running on YARN would have fewer executors for running tasks than the other two cluster managers. Nevertheless, on all the configurations tested for dynamic resource allocation experiments, the Mesos cluster manager produces the best results. The difference in makespan between Standalone and Mesos is small, however, on the configurations with 64 MB and 32 MB RDD partition sizes. With 128 MB RDD partition size, the Mesos cluster manager took 11% and 19% less time when compared with the makespan results observed on Standalone and YARN respectively. The logs generated by Spark shows that for 128 MB configuration on the Standalone, a single task in stage 12 of the *imageClustering_job_2* application was scheduled on one of the slower nodes in the cluster. This task took twice the median time of all the tasks in the stage to complete execution. The substantial difference between the makespan on Standalone and Mesos appears to be caused by the longer time the task took to execute. It is expected that enabling speculative execution of tasks would detect slow running tasks such as this one and schedule them on other available executors. From the results and discussion of the two resource allocation modes above, the following observations can be made:

1. In general, decreasing Spark RDD partition size reduces the makepan and improves the performance of the workload applications regardless of the cluster manager and resource scheduling and allocation policy used, due to increase in the number of concurrently running tasks per application.

2. The time individual jobs spend waiting for resources from the cluster manager substantially contributes to the total makespan of executing all jobs. In addition, slow running tasks in a stage can increase the

total execution time of an application.

3. When few applications (with very minimum resource requirements) within a workload take considerably longer to finish than the rest, it might be necessary to allow the submission of new set of applications into the system once utilization drops below a pre-defined threshold in order to keep the cluster busy.

4. Dynamic resource allocation provides better resource scheduling and allocation when multiple applications need to share the cluster resources. If only one workload application needs to run on the cluster at a time, pre-acquisition of resources using static resource allocation might provide better performance, since applications do not need to acquire the resources at runtime. In this type of scenario, Mesos provides the best performance for efficiently scheduling applications as the size of Spark RDD partition size decreases, followed by Standalone, since they both allocate all available resources to the application. On the other hand, YARN has the worst performance as the number of concurrently running tasks increases, due to allocating only 3 executors to applications by default. In a more realistic production environment, however, usually multiple applications need to share the cluster resources. Using static resources allocation in a production environment introduces many resource scheduling problems, which leads to inefficient cluster resource utilization and increases in makespan.

4.5 Impact of Speculative Execution of Tasks on Makespan

Spark jobs are broken down at run time into several stages that run sequentially. Each of these stages consist of a number of tasks that execute in parallel. This model of execution is sensitive to slow running tasks. This means even a single task taking a longer time to complete can slow down the overall execution of a job. Spark provides the speculative execution of task feature in order to detect such slow running tasks and duplicate them on different executors to speed-up execution. Although this feature is useful on a heterogeneous cluster consisting of slow nodes, for example because of slow CPU (such as the testbed cluster used in this thesis), the speculative execution of task feature of Spark is disabled by default. Therefore, a set of experiments was done to investigate the effects of enabling the speculative execution on jobs makespan.

The experimental setup and runs are the same as in Section 4.4, except speculative execution of tasks was enabled. All the associated configuration parameters related to speculative execution were set to their default values, including *spark.speculation.interval*, *spark.speculation.multiplier* and *spark.speculation.quantile*. The description of these parameters and their default values are provided in Table 2.1. With the default values, after 75% of all tasks in a particular stage completed, Spark would check for tasks to speculate every 100 ms. Any task that is 1.5 times slower than the median of all tasks completed thus far within that stage will be re-launched.

The testbed cluster used for the experiments in this thesis includes two slow nodes with slightly older and slower CPU models, and experiments showed that often these nodes take longer to complete the tasks

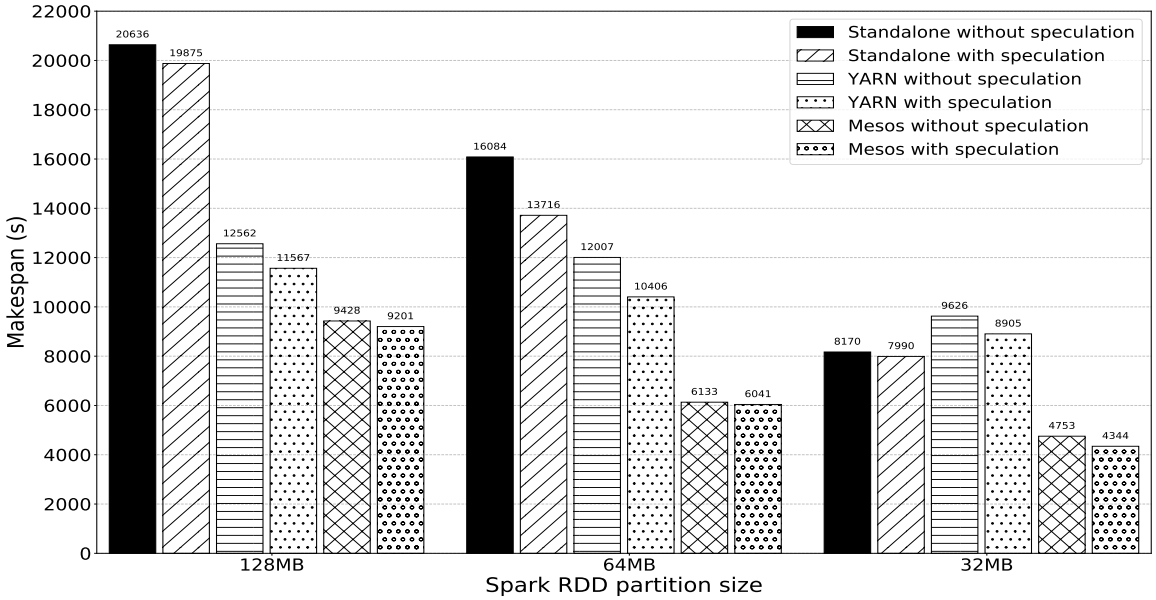


Figure 4.13: Makespan comparison: static with and without speculation/1 core/2 GB

allocated to them compared to the rest of the nine nodes. It was expected that enabling speculative execution of tasks should speed up applications execution and substantially reduce makespan.

Figure 4.13 shows the makespan comparison results for static resource allocation with and without speculative execution of tasks enabled. Surprisingly, a small reduction in makespan values were observed from the figure, especially on the configuration with 32 MB RDD partition size, which has a large number of concurrently running tasks, therefore, higher chances of speculation. The results shows that the reduction in makespan on this configuration is approximately 2% on Standalone, 7% of YARN and 9% on Mesos. Similar results were observed for dynamic resource allocation as shown in Figure 4.14. For example, on the configuration with 32 MB RDD partition size, the reduction in makespan is approximately 9% on YARN and less than 1% on Mesos. These values are lower than expected.

A closer look at the job logs revealed that the low reduction in makespan while using speculation is related to the characteristics of the benchmark image processing applications used for the experiments. Many stages in the applications have long running tasks. Since re-launching tasks means starting them from the beginning on a different executor, the original tasks often finished faster than the speculated tasks and Spark used the results of the tasks that finished first. In addition, Spark sometimes re-launches the speculated tasks on executors hosted on the same (slow) node as the straggler tasks. In this scenario, the straggler tasks often complete before the speculated tasks.

Completely unexpected results were observed, however, on the Standalone cluster manager when the RDD partition size was set to 32 MB. Figure 4.14 shows that speculating the tasks for this configuration increased

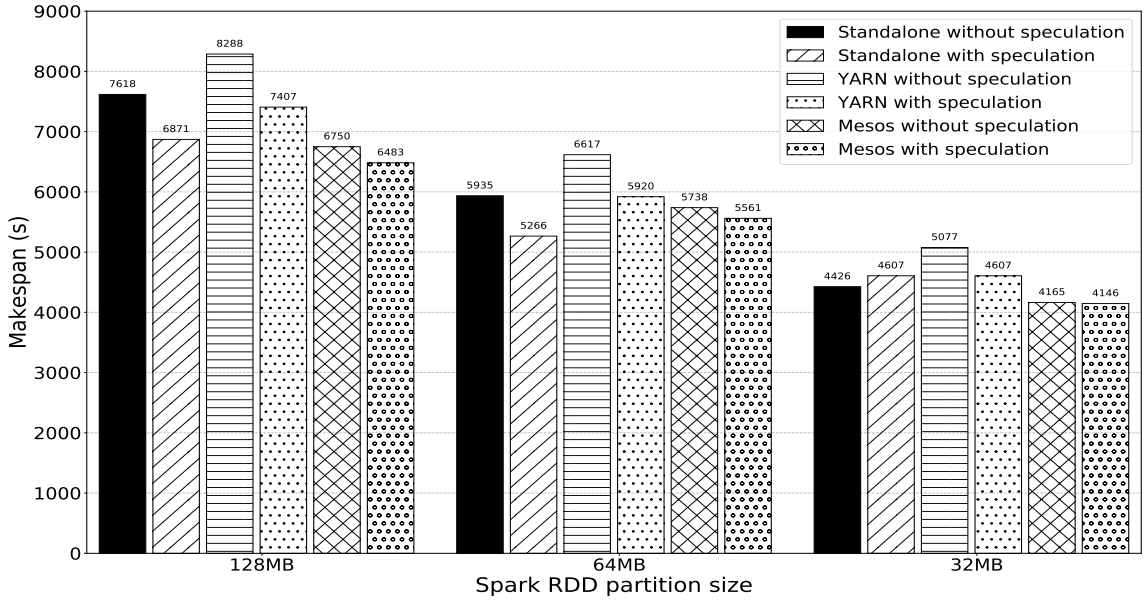


Figure 4.14: Makespan comparison: dynamic with and without speculation/1 core/2 GB

the makespan by approximately 4%. The log file revealed the increase in makespan on this configuration is due to an instance of *imageClustering_job_3*. Unlike all other instances of the Image Clustering application which spend the highest amount of time in stage 12, the *imageClustering_job_3* strangely spent large amount of time in stage 1. Further analysis shows that this stage has 27 tasks of which 5 are stragglers and were re-launched on different executors. Out of the 5 speculated tasks, only 1 succeeded in completing whereas the other 4 were killed by Spark because the initial (slow) tasks finished first. Interestingly, among the straggler tasks, one was running on *mario* and later re-launched on *luigi*: one of the two slow nodes in the cluster. Both the initial and the speculated tasks took a long time to execute compared to the rest of the tasks in the stage. The tasks in stage 1 of the *imageClustering_job_3* instance have a median running time of 1 second and 75^{th} percentile of 4 seconds. However, the straggler task running on *mario* took 9 minutes to complete. Furthermore, since the straggler task was speculated on a slow node, the initial task eventually completed before the speculated task. This contributed to the high makespan value observed for this configuration.

4.5.1 Analysis of Speculation on the Workload Applications

Impact of Speculation on the Image Clustering Application

The three instances of the Image Clustering application spent the highest amount of time in stage 12. When the RDD partition size is set to 128 MB on the Standalone, the log files of the *imageClustering_job_1* and *imageClustering_job_2* shows that each of these two instances of the application has two slow running tasks

in stage 12, one of the slow tasks was scheduled on *mario* and the other on *luigi*. These slow tasks took approximately twice the median running time of the all tasks within the stage. For the *imageClustering_job_1* instance, Spark restarted the two slow running tasks on different executors hosted on the same nodes (one on *mario* and the other on *luigi*). The straggler tasks of the *imageClustering_job_2* instance were speculated on *worker8* and *worker9* nodes. Similarly, the log file of the *imageClustering_job_3* instance shows that one task in stage 12 took approximately twice the median of all the tasks in the stage to execute on *mario*. This task was later speculated on a different Spark executor hosted on *worker1*.

The tasks in stage 12 of the Image Clustering application are long running, therefore, none of the speculated tasks succeeds in completing before the straggler tasks, since it requires a large amount of time to re-start the tasks from the beginning, forcing Spark to kill the speculated tasks. This application exhibited similar characteristics on other configurations (that is static and dynamic resource allocation with 128 MB, 64 MB and 32 MB RDD partition sizes) on the three cluster managers.

Figure 4.15 shows the running times of tasks in stage 12 of the Image Clustering application when the RDD partition size is set to 32 MB. Since the running time distributions for this application observed on all the configurations exhibit similar properties, only the result of Standalone cluster manager with 32 MB RDD partition size is shown. The *Executor Wait Time* on the graph includes the time executors spend de-serializing tasks, fetching data, serializing results and executing garbage collection, while the *Executor Run Time* is the time executors spend running tasks. In general, the *Executor Wait Time* is very small in comparison to the *Executor Run Time*.

On the configurations with 32 MB and 64 MB RDD sizes, there are more tasks within each stage. These tasks process less data compared to the configuration with 128 MB size, therefore, require less time to complete. This results in having more speculation done. For example, stage 12 of the Image Clustering application has a total of 39 tasks when the partition size is set to 32 MB compared to 10 tasks on the configuration with 128 MB size. Among the 39 tasks, 4 tasks (2 on *mario* and 2 on *luigi*) were speculated on either *luigi*, *worker7* or *worker9*. None of the speculated tasks, however, succeed in completing before the stagger tasks.

Impact of Speculation on the Flower Counter Application

There are multiple stages that contribute to the execution time of the Flower Counter application. The log file of the configuration with 128 MB RDD partition size shows that the stages in all the three instances of this application exhibit similar properties. Almost all the stages in the *flowerCounter_job_2* and *flowerCounter_job_3* that took a large amount of time to execute have one slow task (that took approximately twice the median running time of all tasks in the stage) running on either *mario* or *luigi*. These slow tasks were speculated on different cluster nodes but did not succeed in completing before the original straggler tasks, causing Spark to kill the speculated tasks.

On the *flowerCounter_job_1* instance, two stages have one slow running task each, scheduled respectively

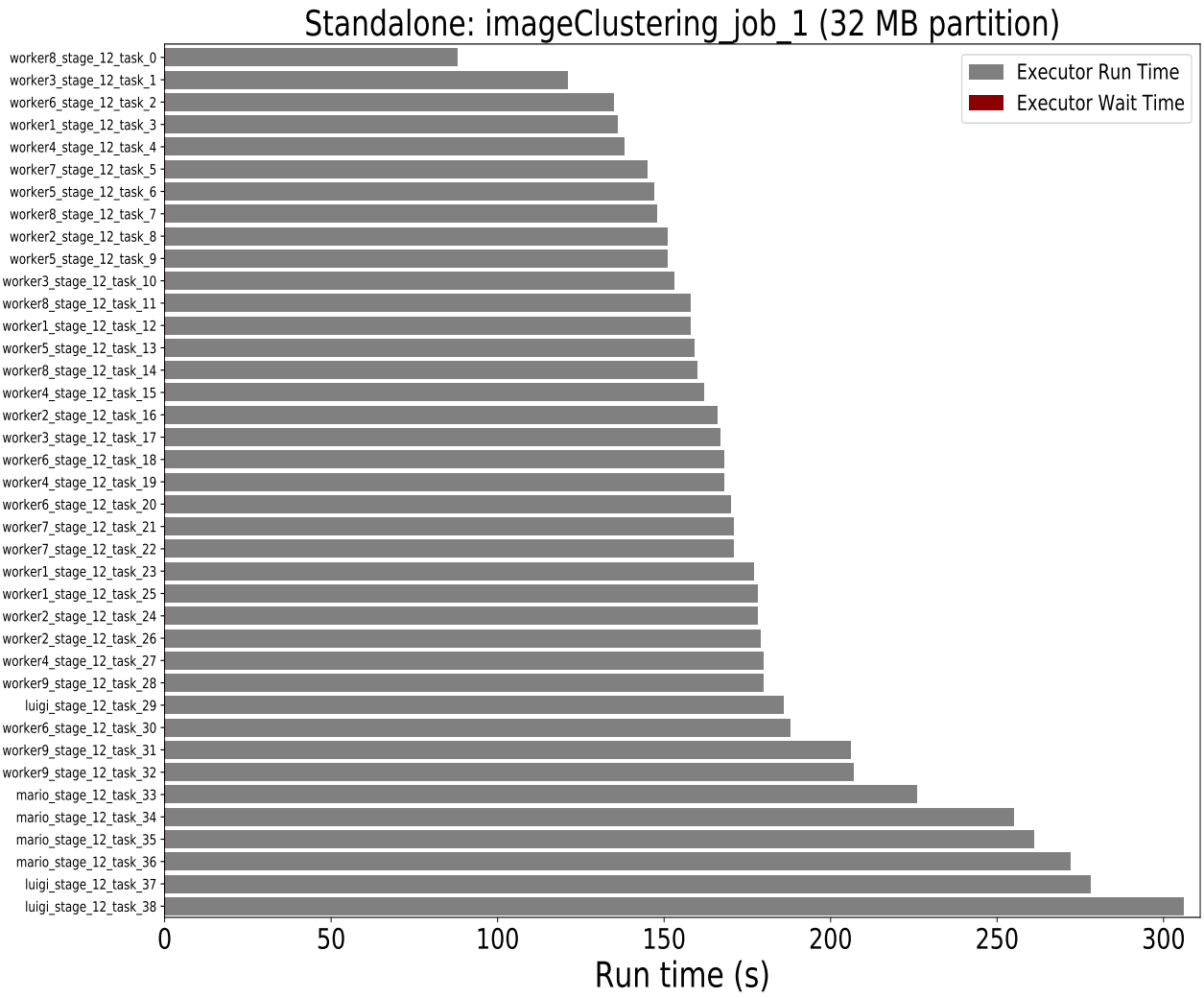


Figure 4.15: Tasks running times of the Image Clustering application: Standalone/static/1 core/2 GB/with speculation

on *worker1* and *worker9*, and eventually succeeded in completing before the straggler tasks. Spark used the results of the speculated tasks to proceed to the next stage in the application execution graph. Interestingly, there are stages in the Flower Counter application where speculation is not activated despite uneven task running times distribution. For example, stage 5 of the *flowerCounter_job_3* has four tasks. These tasks were scheduled on *worker1*, *worker2*, *mario* and *luigi*, and respectively took 1.5 minutes, 3.5 minutes, 7.1 minutes, 7.1 minutes to execute. The default values used for the parameters associated with speculative execution of tasks prevent the tasks in this stage from being speculated, since 75% (that is 3 tasks in this case) of all the tasks within a giving stage must complete before speculation is activated for the particular stage. All instances of the Flower Counter application exhibit similar properties on other configurations.

Figure 4.16 shows the running times of tasks in seven stages of the Flower Counter application that took the highest amount of time to complete when the RDD partition size is set to 32 MB. Similar to the Image

Clustering application, as the RDD partition size gets smaller, the chances of having straggler tasks increases due to more concurrently running tasks. For example, with 32 MB RDD partition size, there are 10 tasks in stage 5 of the *flowerCounter_job_1* instance, 3 of which are stragglers (2 on *luigi* and 1 on *mario*). These tasks were later speculated on *worker4*, *worker5* and *worker9*. Only 1 out of the 3 speculated tasks succeed, however, in completing before the straggler task.

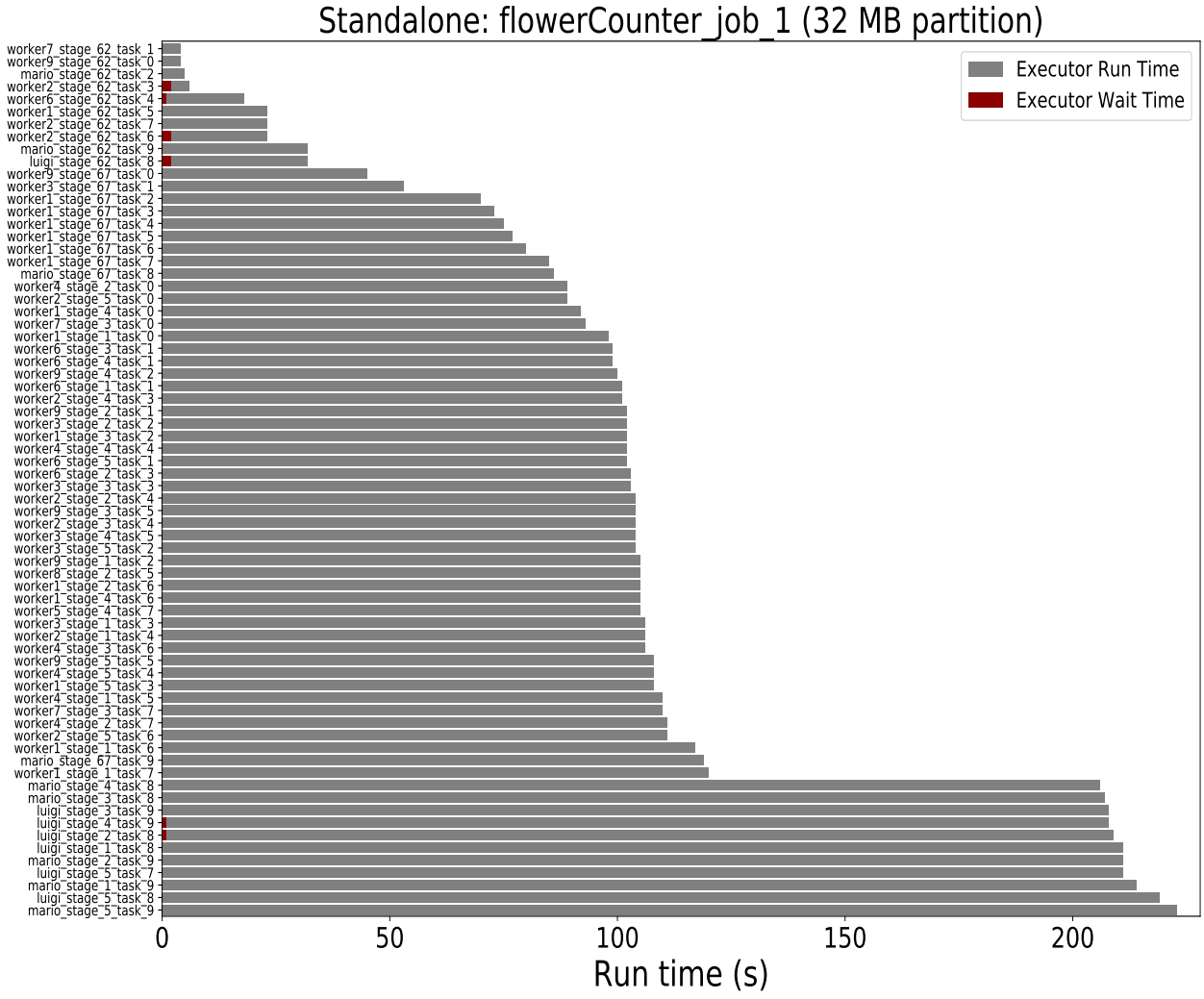


Figure 4.16: Tasks running times of the Flower Counter application: Standalone/static/1 core/2 GB/with speculation

Impact of Speculation on the Image Registration Application

The three instances of the Image Registration application spend the highest amount of time in stage 2. On the configuration with 128 MB RDD partition size on Standalone, there are no slow running tasks and no speculation is done in any of the three instances of this application. For example, the tasks in stage 2 of the *imageRegistration_job_1* has a median running time of 9.1 minutes, 75^{th} percentile of 13 minutes and

maximum running time of 15 minutes. Figure 4.17 shows the running times of tasks in stage 2 of the Image Registration application when the RDD partition size is set to 32 MB on Standalone. This application shows similar tasks running times characteristics on other configurations.

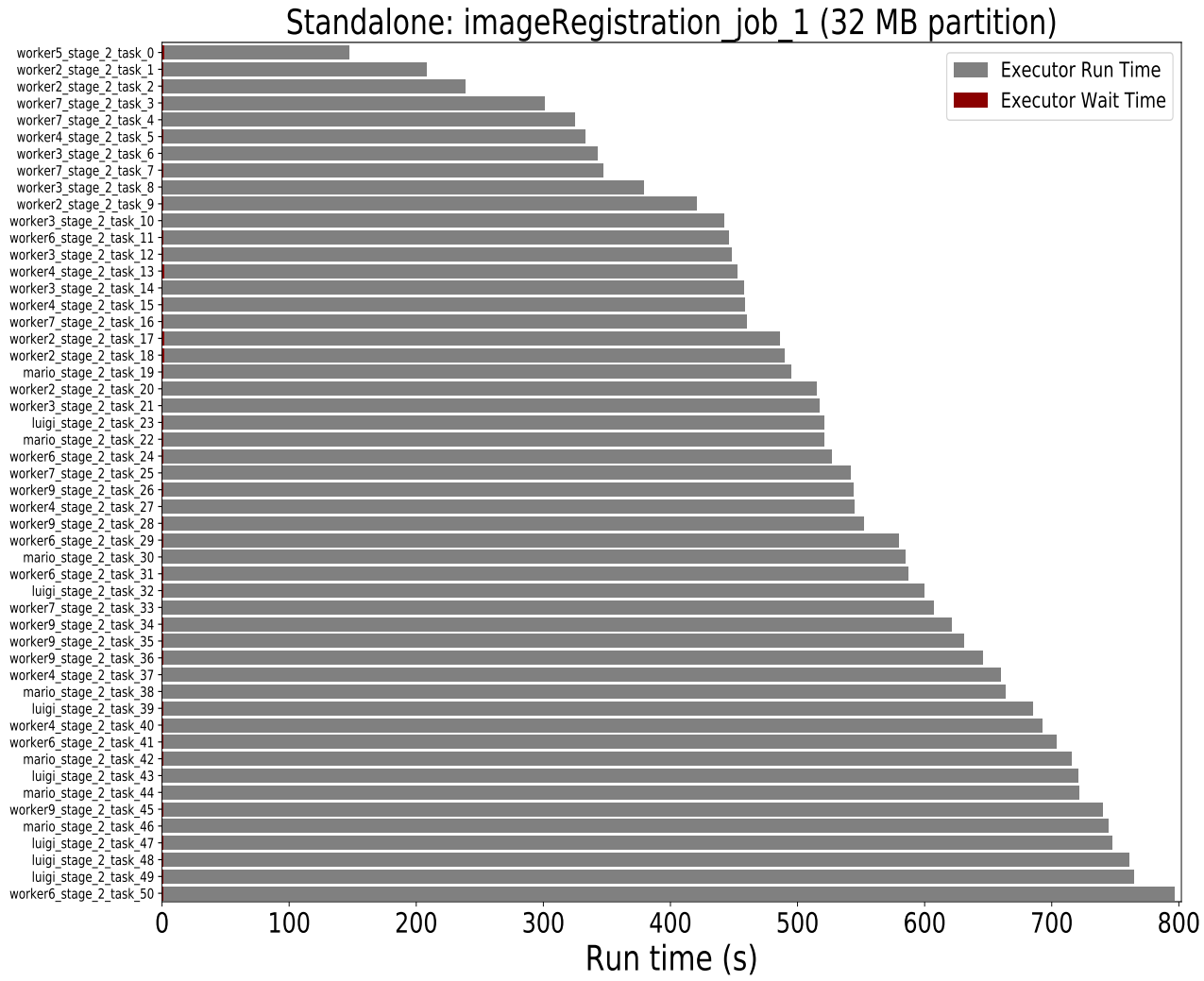


Figure 4.17: Tasks running times of the Image Registration application: Standalone/static/1 core/2 GB/with speculation

Similar to the two other workload applications discussed above, there are some configurations in which speculation was activated in stage 2 of this application, such as *imageRegistration_job_1* on the configuration with 128 MB RDD partition size on Mesos. In this instance of the application, stage 2 has 14 tasks, of which 3 are stragglers (2 on *mario* and 1 on *worker2*). The straggler tasks were speculated on *worker1*, *worker4* and *worker9*. Only 1 out of the 3 speculated tasks succeed in completing before the straggler.

The discussion above shows that there are multiple considerations when enabling speculation. The various configuration parameters related to speculation play an important role in the effectiveness of the feature. Setting high values for the parameters that control how slow a task should be before it can be considered for

speculation and the percentage of tasks that must complete in a stage before speculation is activated would reduce the number of tasks to speculate and minimize the chances of having the speculated tasks succeed in completing before the straggler tasks, especially in applications with long running tasks (such as the image processing applications used in this thesis). On the other hand, setting very low values for these parameters could result in an aggressive speculation, which may eventually increase the latency of jobs.

Therefore, further experimentations and additional applications are needed to be able to draw any conclusion on the validity of speculative execution for these type of applications. Perhaps the values which trigger speculation are set inappropriately for the experiments in this section. Finding the optimal values for the different parameters associated with speculative execution of tasks that would provide the best performance for the image processing applications used is outside the scope of this thesis.

4.6 Impact of Spark Executor Size on Makespan

The two main resources that Standalone, Mesos and YARN manages are CPU cores and memory. Although other resources such as disk and network I/O could affect Spark application performance as well, only CPU and memory resources are considered for evaluation in this thesis. Every Spark executor in an application has the same fixed number of cores and the same fixed heap size controlled by *spark.executor.cores* and *spark.executor.memory* configuration parameters respectively. In addition to the amount of data that can be stored in RDDs, *spark.executor.memory* determines the maximum sizes of the shuffle data structures used for operations such as *grouping*, *aggregations*, and *joins*.

A number of experiments were done to evaluate the performance impact of Spark configuration parameters (that control executor size) on the workload applications used in this thesis. The results reported for the configuration with *1 CPU core/2 GB memory* executor size were taken from the experiments in the previous sections. For the remainder of this section, Spark executor size refers to the number of CPU cores and memory of the executor. In addition to testing the three cluster managers with static and dynamic resource allocation, the following configurations were used for the experiments in this section.

- Spark executor size: 3 CPU cores/6 GB memory, 6 CPU cores/12 GB memory,
- Spark RDD partition size: 128 MB, 64 MB, 32 MB,
- Speculative execution of tasks: Disabled.

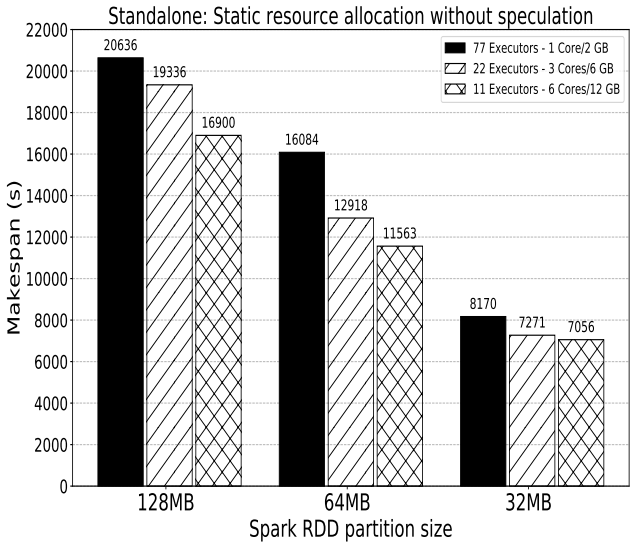
The makespan results for various experimental configurations on the Standalone cluster manager are shown in Figure 4.18. On all the three configurations using static resource allocation without speculation, the makespan value decreases as the size of RDD partitions were decreased due to increase in number of concurrent tasks as shown in Figure 4.18a. In addition, the figure shows that the makespan decreases with the increase in the number of CPU cores and memory per executor. For example, with 128 MB RDD partition size, the makespan reduced by 6% when the executor size is increased from *1 CPU core/2 GB memory* to *3*

CPU cores/6 GB memory, and 16% from *3 CPU cores/6 GB memory* to *6 CPU cores/12 GB memory*. A similar reduction in makespan is noticed when the size of the RDD partition is set to 64 MB and 32 MB. For the configuration with 32 MB RDD partition size, however, the reduction is small between *3 CPU cores/6 GB memory* and *6 CPU cores/12 GB memory*.

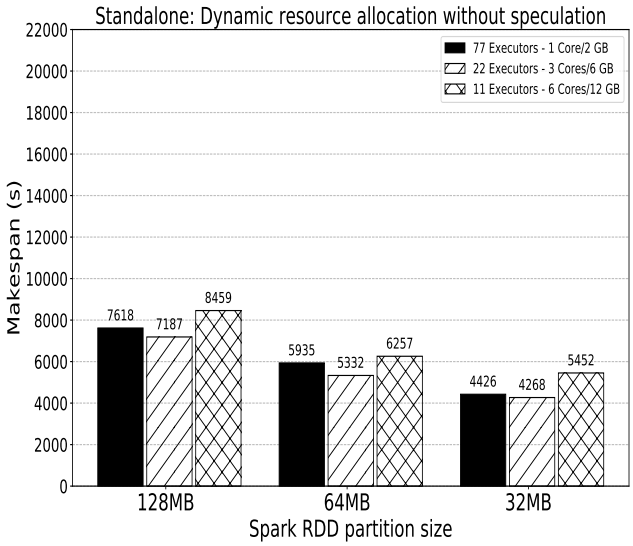
Spark executors are instances of JVMs, consisting of a number of CPU cores and memory. Each core within an executor runs a single task at a time. Therefore, setting Spark executors with many CPU cores has the benefit that multiple tasks can run in a single JVM and share memory space. Applications that use broadcast variables tend to benefit from shared memory space. Broadcast variables are shared read-only variables cached on Spark worker machines, and are used to provide nodes with a copy of a large dataset using efficient broadcast algorithms to reduce network communication cost. Spark automatically broadcasts data needed by tasks within each stage. The data broadcasted this way is cached in serialized form and de-serialized before running each task. Executors with a small amount of resources means more JVMs on the worker machine. This may increase the overhead of creating many small JVMs and may lead to performance deterioration for applications that use broadcast variables, because the broadcast variables must be replicated on each executor. Even if the application programmer did not explicitly define broadcast variables within their applications, Spark could still take advantage of broadcast variables (internally) during shuffle to share data needed by tasks within a stage. This is why the experimental setup with *1 CPU core/2 GB memory* generally provides the worst performance.

Interestingly, when dynamic resource allocation is enabled on the Standalone cluster manager without speculation, the configuration with *6 CPU cores/12 GB memory* provides worse performance on all the three Spark RDD partition sizes. Figure 4.18b shows an increase in makespan of approximately 15% on the configurations with 128 MB and 64 MB, and 22% on the configuration with 32 MB RDD partition sizes, when the executor size is increased from *3 CPU cores/6 GB memory* to *6 CPU cores/12 GB memory*. Remember dynamic resource allocation works by allocating executors to applications based on the current workload, and removes executors only when they are idle. This means setting executors with *6 CPU cores/12 GB memory* would result in having few executors to allocate to applications compared to the other two configurations. For example, stage 2 of the *imageRegistration_job_1* instance has 14 tasks. With *6 CPU cores/12 GB memory* executor size, this application would request 3 executors from the cluster manager. Only 2 of the 3 allocated executors may be fully utilized by the application, however, while the remaining 1 may only run 2 tasks. This means 4 CPU cores would be completely idle during the execution of this application. The cluster manager can only reclaim the resources when the two tasks have finished executing. This concept contributes to the increase in makespan values observed on the configurations with *6 CPU cores/12 GB memory* executor size.

Similar to the results observed for the Standalone, the configuration with *6 CPU cores/12 GB memory* provides the best performance results on the YARN cluster manager when using static resource allocation configurations without speculative execution of tasks as shown in Figure 4.19a, except for 64 MB RDD partition size where the makespan increased by approximately 15% when the executor size is increased from



(a) Makespan: Standalone/static/no speculation



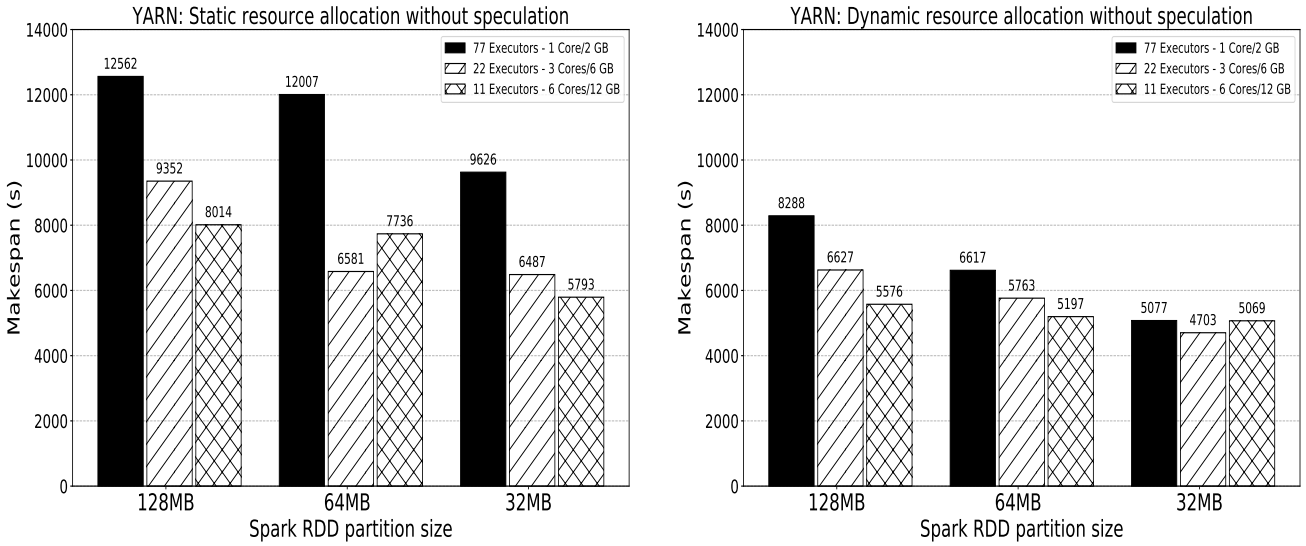
(b) Makespan: Standalone/dynamic/no speculation

Figure 4.18: Makespan of various executor sizes on Standalone without speculation

3 CPU cores/6 GB memory to *6 CPU cores/12 GB memory*. With 64 MB RDD partition size, applications on average spend 39 seconds waiting for resource on the configuration with *3 CPU cores/6 GB memory*, and 1044 seconds on *6 CPU cores/12 GB memory*. This shows there is approximately 16 minutes increase in waiting time when the executor size is increased from *3 CPU cores/6 GB memory* to *6 CPU cores/12 GB memory*.

A closer look at the log file shows that on the configuration with *6 CPU cores/12 GB memory*, a substantial number of tasks were scheduled on *mario* and *luigi*. Since speculation is not used for these configurations, the running times distribution of tasks within the stages are uneven. For example, stage 3 of the *flower-Counter_job_3* has 7 tasks. Three out of the 7 tasks were scheduled on *worker4* and completed on average in 2 minutes, while the remaining 4 tasks were scheduled on *mario* and completed on average in 18 minutes. This shows 16 minutes difference in tasks running times within the same stage depending on the node the tasks are scheduled to run. Ideally, enabling speculative execution for stages such as this one should substantially reduce the running time. Speculative execution is not tested for these configurations, however, because the results of speculation on *1 CPU core/2 GB memory* in Section 4.5 shows there are still uneven task run-time distribution when using speculative execution as a result of the two slow nodes in the testbed cluster, and the values set for the various parameters associated with speculation.

Similarly, the *6 CPU cores/12 GB memory* executor size provides the best performance results when using dynamic resource allocation, except on the configuration with 32 MB RDD partition size where the makespan increased by approximately 7% as shown in Figure 4.19b. The log file shows that many of the tasks scheduled on *mario* and *luigi* took longer time to complete compared to the tasks in the same stage



(a) Makespan: YARN/static/no speculation

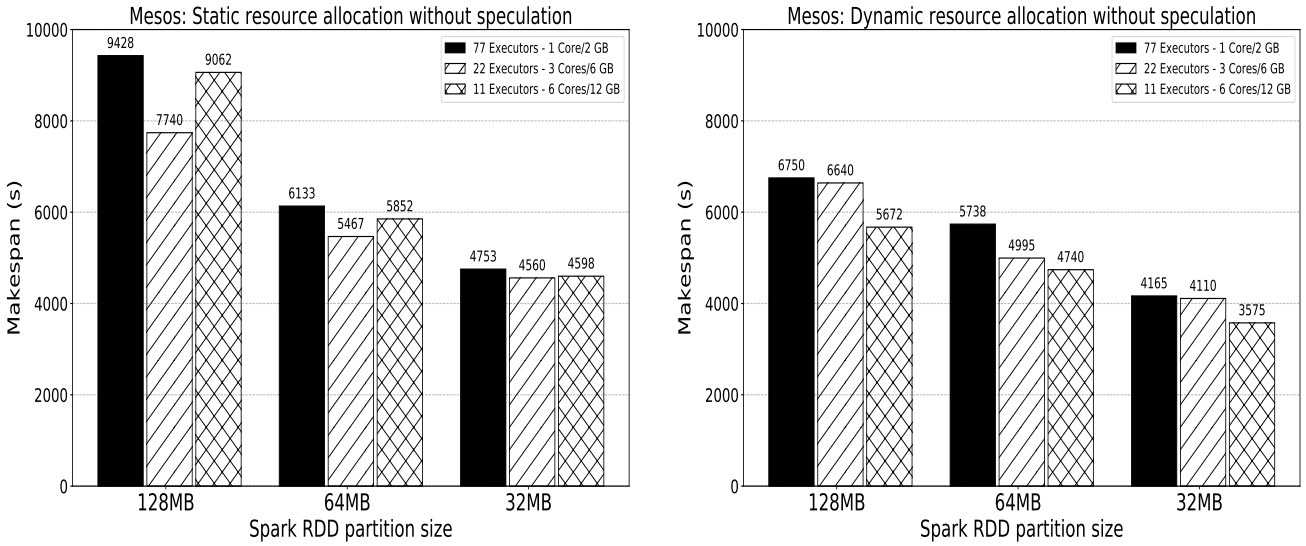
(b) Makespan: YARN/dynamic/no speculation

Figure 4.19: Makespan of various executor sizes on YARN without speculation

scheduled on other cluster nodes. In some stages, all the tasks are scheduled on one (or both) of the slow nodes, which contributes to the increase in makespan.

Figure 4.20a compares the makespan of various executor sizes on the Mesos cluster manager using static resource allocation without speculation. Strangely, when the executor size is increased from *3 CPU cores/6 GB memory* to *6 CPU cores/12 GB memory*, the makespan increased as well. As the partition size gets smaller (such as 32 MB), however, the difference in makespan between *3 CPU cores/6 GB memory* and *6 CPU cores/12 GB memory* becomes small. The log file for the *6 CPU cores/12 GB memory* configuration shows that during the execution of some stages (such as stage 4 of the *imageClustering_job_2* instance), some executors exited with *Out of Memory Error* due to Mesos containers (running the executors) exceeding their allocated resource threshold. All tasks running on the lost executors failed and were restarted by Spark on different executors. In addition, the uneven tasks run-time problem explained earlier was also noticed in this configuration and appears to contribute to the increase in makespan of both 128 MB and 64 MB RDD partition sizes. As expected, when dynamic resource allocation is used without speculation, the *6 CPU cores/12 GB memory* executor size configuration shows the best performance results on all the three RDD partition sizes as shown in Figure 4.20b.

From the above results, it is clear that the choice of resource size per Spark executor impacts the performance of applications. In general, the optimal balance of cores and memory per executor varies per workload. While more cores per executor reduces per-executor overhead, there are other considerations which affect performance negatively with an increase in the number of cores per executor. For example, it could lead to contention problems on shared clients like the HDFS and shared data structures which require synchroniza-



(a) Makespan: Mesos/static/no speculation

(b) Makespan: Mesos/dynamic/no speculation

Figure 4.20: Makespan of various executor sizes on Mesos without speculation

tion. This is because every CPU core within an executor reads and writes its own data separately. Having many concurrent threads (reading and writing to HDFS) could result to inefficient I/O throughput. In addition, too much memory per executor process (which is a JVM) often results in excessive garbage collection delays. This is not a problem for the workload applications used in this thesis, however, since the applications are mostly CPU intensive and do not make a substantial use of HDFS during execution. The only stages the applications communicate with HDFS are when reading the input file (that is the first stage) and when writing the final output (which is a small text file) during the final stage.

Uneven tasks running times within the same stage is a more serious problem that affects the makespan of the workload applications when using *6 CPU cores/12 GB memory* per Spark executor, as a result of having slow machines as part of the cluster nodes. Executor failure and task re-submission due to exceeding resource allocation threshold also contribute to the increase in makespan of the applications.

Therefore, there are many factors to consider in deciding the optimum numbers of CPU cores and memory per Spark executor. Applications with multiple stages that require shuffle will benefit from having executors with large resource size. However, the number of CPU cores and memory chosen per Spark executor should not be too large (in relation to the total amount of resources on the node) to avoid resource contention, excessive garbage collection and executor failure which may negatively affect applications performance. Moreover, for applications with long running tasks within stages, having large number of resource per executor may affect performance, especially on a heterogeneous cluster.

4.7 Analysis of Makespan Performance Improvements

One of the main motivation of this thesis is to parallelize a set of image processing applications in order to minimize their processing time. In Section 4.2, makespan results of running the sequential version of the applications on a large server (called *Onomi*) were compared with the results of processing the same dataset using Spark (with default parameter settings). This section compares the same results from *Onomi* with the results of a particular Spark configuration. The goal is to show that parallelizing these sets of applications, as well as tuning various Spark configuration parameters could substantially reduce the makespan.

In the previous sections, the three Spark cluster managers tested provided the best makespan results when using dynamic resource allocation. Therefore, dynamic resource allocation with 1 core and 2 GB memory per executor, 32 MB RDD partition size, and without speculative execution of tasks was selected for comparison in this section.

Recall that *Onomi* has 56 virtual cores, while the Spark cluster is equipped with 77 virtual cores. Therefore, to provide fair comparison between the two testbeds, all Spark results were multiplied by the total number of virtual CPUs on Spark divided by the total number of virtual CPUs on *Onomi*. For example, if Spark processed the workload with 77 CPU cores in 4426 seconds, the same workload will supposedly take approximately (77/56)*4426 = 6086 seconds to execute on Spark with 56 virtual cores if the applications scale equally with CPU cores only.

Figure 4.21 compares the makespan results of *Onomi* and the various Spark cluster managers. The figure shows the makespan of the applications was reduced by approximately 63% on the Standalone, 57% on YARN and 65% on Mesos cluster managers when compared with the results from *Onomi*. Although *Onomi* is a relatively newer machine compared to the nodes of the testbed cluster, a substantially better performance could be achieved by simply distributing the processing across multiple commodity machines using Spark and tuning the various Spark configuration parameters.

In addition to achieving a better performance in terms of makespan, Spark provides other advantages over a traditional large server:

- Scalability: As data sources become more diverse and new applications are deployed, there is the need to increase both storage and computing power to keep up with new processing requirements. Unlike traditional large servers, Spark is highly scalable. It is capable of processing very large datasets across hundreds of inexpensive computers in parallel.

- Resilience to failure: A key advantage of using Spark is its high availability and resilient for failure feature. It supports both applications with high reliability and availability requirement such as long-running services that should always be available and short-lived latency-sensitive requests servicing end-users.

- Cost effective: It is often cost effective to build a large cluster of commodity machines for both dis-

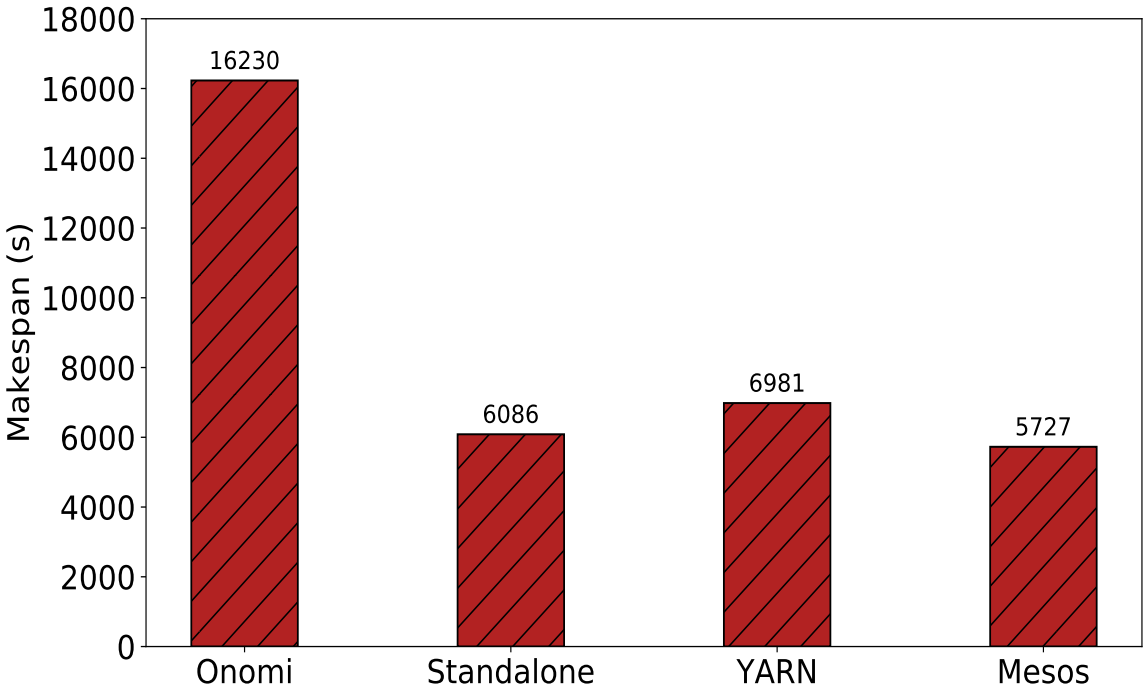


Figure 4.21: Makespan of the workloads on *Onomi* and Spark with various parameter tuning

tributed storage and processing of large dataset compared to traditional servers which are often difficult and costly to scale in order to keep up with the requirements of storage and processing of massive volumes of data. For example, each of the 12 nodes in the Spark cluster cost approximately $1,400. The entire cluster (with one master and eleven worker nodes) would cost approximately $16,800 to build. The cluster has networking hardware requirements, and additional management costs over the lifetime of the system. The network switch could be purchased for less than $200, making the total cost of the cluster around $17,000. Similarly, *Onomi* cost approximately $40,000, which is 2.4 times the cost of the Spark cluster. This shows Spark can achieve better performance with substantially less cost in the scale of a small cluster versus a single machine. In addition, as the requirements for more processing power arises, simply adding more nodes to the Spark cluster would increase the number of cores, and hence the processing power. On the other hand, it is costly and challenging to add more cores to an already built server such as *Onomi*.

4.8 Summary

This chapter presented the performance results of processing multiple image processing applications on Spark. Various cluster managers that support Spark where evaluated under different Spark parameter settings.

In addition, an analysis was presented on the impact of Spark parameters on makespan. Based on the experiments conducted and the results presented in this chapter, it is shown that distributed processing of multiple applications on a Spark cluster of commodity machines can substantially improve performance and minimize makespan compared to sequential processing using a specialized server equipped with large amount of resources.

Furthermore, the results in this chapter prove that the default parameter settings of Spark are not necessarily efficient for all type of applications. In particular, the various image processing applications considered in this thesis show a worse performance in terms of cluster resource utilization and applications makespan when using default values for the various Spark parameters tested, including the parameters that control resource scheduling and allocation, number of parallel tasks per job and resource size per executor.

Therefore, even though Spark generally provides better performance than traditional servers, to share cluster resources efficiently between multiple applications and achieve better cluster resource utilization, various Spark configuration parameters need to be adjusted in order for the applications to take advantage of all the compute power, memory and networking capacity of the cluster.

In conclusion, it is expected that the insights learned from the case studies will be helpful for both cluster administrators and users to understand the impact of various Spark configuration parameters on sample image processing workloads in terms of application's performance and cluster resource utilization.

Chapter 5

Conclusions

Apache Spark is a widely used in-memory framework for large-scale distributed data processing and analytics on a cluster of computers. Spark applications can run on various cluster managers including Spark Standalone, Apache Mesos, and Hadop YARN. In addition, Spark provides many configuration parameters to control scheduling and resource management across multiple applications. However, in order to share cluster resources efficiently, the multiple applications submitted to the cluster must be appropriately scheduled to take advantage of all the compute power, memory and networking capacity of the cluster. Despite the wide adaptability of Spark in academia and industry, little is known about the effect of various Spark configuration parameters on image processing applications performance. This thesis helps in understanding the characteristics of Spark configuration parameters and how they affect cluster resource utilization and jobs makespan. The remainder of this Chapter contains a summary of the thesis, and presents discussion on the thesis contribution as well as suggestions on possible future work.

5.1 Thesis Summary

There is not much published literature that provide in-depth analysis of the performance implications and overhead of various Spark scheduling and resource allocation configuration parameters on image processing applications. This leads to performance issues, because cluster administrators do not know which cluster manager to use for managing their cluster, what mode of resource allocation to use, and how to configure the multiple scheduling and resource allocation parameters of Spark in order to achieve efficient cluster resource utilization. Furthermore, cluster users must understand how to tune the parameters of Spark to get the best application performance. Therefore, it is very important to provide extensive analysis of how the different configuration parameters as well as resource allocation policies implemented in the three Spark cluster managers affect performance. This would help cluster administrators and users choose the best cluster manager for their image processing applications, and understand the exact effect of the different parameter settings in terms of resource utilization and jobs makespan.

This thesis demonstrates the feasibility of using Apache Spark for distributed processing of large amount of image data in a time and cost efficient manner. A set of heterogeneous image processing applications including Image Registration, Flower Counter and Image Clustering were parallelized. An extensive analysis

and comparison was presented to evaluate the performance of running these applications on a large server and a Spark cluster using three different cluster managers for resource allocation including Spark Standalone, Apache Mesos and Hadoop YARN. In addition, the performance implications as well as overhead of the two different job scheduling and resource allocation modes available in Spark i.e. static and dynamic resource allocation were examined. Furthermore, the thesis explored the various configurations available on Spark that controls speculative execution of tasks, resource size and number of parallel tasks per job, and explained their impact on image processing applications.

5.2 Thesis Contributions

This thesis explored distributed image processing on Spark, and provided an in-depth analysis of how to minimize the makespan of multiple applications and improve cluster resource utilization through configuring various Spark parameters. The thesis makes the following contributions:

1. The thesis suggested using Spark as a viable means of distributing and running multiple image processing applications on a cluster. A set of heterogeneous applications written in Python were converted into Spark applications using the *PySpark* API to show that Spark is capable of large scale distributed processing of image data. Various Spark transformations and actions such as *map*, *filter*, *aggregate*, as well as optimized and distributed machine learning algorithms such as *K-Means* clustering were used to replace the sequential version provided by Python.

2. The work in this thesis provided an in-depth analysis of the performance implication as well as potential overhead of the two Spark resource allocation modes: static and dynamic, and the three Spark supported cluster managers: Standalone, Mesos and YARN, and explained the effect of the various configuration parameters on applications performance. Furthermore, the thesis used three domain specific image processing applications for evaluation instead of the generic big data benchmarks that are commonly used in most of the previous works.

3. Through experimentations, the thesis showed that Spark configuration parameters such as number of parallel tasks per job and the size of Spark executor can have a substantial impact on makespan. More importantly, the thesis shows how the various Spark configuration parameters could be configured in order to minimize jobs makespan and ensure that multiple applications submitted to the cluster managers are allocated a fair share of the cluster resources in a time effective manner and are able to run concurrently on a shared cluster.

4. The work in this thesis showed that when static resource allocation is used with the default values of other configuration parameters (such as the parameters that control resource size per Spark executor), it may lead to a very poor cluster resource utilization. Furthermore, the thesis suggested enabling the dynamic resource allocation to improve cluster resource utilization in any production Spark cluster

(regardless of the underlying cluster manager) that is shared by multiple users to run applications concurrently.

5.3 Future Work

There are considerable opportunities for future work. In particular, the following areas were identified:

1. Impact of schedulers on image processing applications: many schedulers have been developed and integrated into cluster managers such as Hadoop YARN and Mesos. These schedulers implement different resource scheduling and allocation policies to solve the problems introduced by the default schedulers provided out of the box by the resources managers. The experiments in this thesis were designed to focus on evaluating the performance impact of Spark configuration parameters on image processing applications. Therefore, the default schedulers provided by each of the three cluster managers tested were used. However, more experiments and analysis can be done to study the impact of various schedulers such as ARIA, PRISM, Starfish, DynMR, Delay Scheduling, Quincy etc., on image processing applications in terms of cluster resource utilization, service level objectives, dynamic tuning and reconfiguration of parameters, data skew, data locality and job completion time.

2. Porting external cluster managers to support Spark applications: support for additional cluster managers are being added to new versions of Spark. For example, during the period of writing this thesis, Spark added support for running applications on clusters managed by Kubernetes.[1] Therefore, in addition to the three cluster managers evaluated in this thesis, other cluster managers such as Quasar, OPERA, Omega, Borg could be ported to support Spark application. Furthermore, experiments could be done to evaluate the performance of Kubernetes and other ported cluster managers (that do not support Spark applications by default) in terms of cluster resource utilization and makespan.

3. Developing an automated cluster resource monitoring system: among the insights gained from this thesis is that workload submission pattern could affect jobs makespan. When the jobs submitted to the cluster manager (using dynamic resource allocation) consist of a mixture of applications with short and long running tasks, some of the jobs will finish earlier than the others, leaving the cluster almost idle at some point in time. However, simply submitting more jobs to the cluster without monitoring the resource utilization could lead to interference between workloads which could result in severe resource contentions and performance degradation. Therefore, a monitoring system could be developed to automatically monitor cluster resource usage and submit a new set of applications when the cluster utilization drops below a certain threshold.

4. Performance of image processing on other big data frameworks: although Spark substantially reduced

[1] `https://kubernetes.io/` (10 Apr, 2018)

the makespan of the applications used in this thesis, experiments could be done to quantify the performance difference between running these sets of applications in Spark versus in other more mature big data frameworks such as Hadoop. This could provide insights as to whether the in-memory feature of Spark is beneficial to these sets of image processing applications.

5. Including other performance metrics in the experimental design and evaluation: the experimental design used in this thesis could be extended to include measuring and analysing the performance impact of Spark configuration parameters on other important metrics such as disk and network I/O throughput. Profiling and logging applications runtime resource utilization such as CPU, memory, network and disk is important in designing a system that could predict computation requirements as well as completion time of applications under various data sizes.

6. Automatic tuning of configuration parameters for broad applications: it is experimentally shown in this thesis that the optimal values for Spark configuration parameters such as speculative execution of tasks and resource size per Spark executor depend on the workload. In addition, Spark features such as speculative execution and dynamic resource allocation has other configuration parameters associated with them. This thesis only tested the effect of enabling or disabling these features on image processing applications. More experiments could be done to understand how to set other parameters associated with speculative execution of tasks as well as dynamic resource allocation in order to improve performance. More importantly, an automatic tuning system could be developed and integrated into the Spark platform to dynamically monitor applications and automatically adopt and reconfigure parameters at runtime in order to improve applications performance.

References

[1] Ablimit Aji, Fusheng Wang, Hoang Vo, Rubao Lee, Qiaoling Liu, Xiaodong Zhang, and Joel Saltz. Hadoop GIS: A High Performance Spatial Data Warehousing System Over MapReduce. *Proceedings of the VLDB Endowment*, 6(11):1009–1020, August 2013.

[2] Amazon EC2. `https://aws.amazon.com/ec2/`. Accessed: 6 March, 2018.

[3] Apache Spark Examples. `https://spark.apache.org/examples.html`. Accessed: 11 January, 2018.

[4] Pablo Arbelaez, Michael Maire, Charless Fowlkes, and Jitendra Malik. Contour Detection and Hierarchical Image Segmentation. *IEEE Transactions on Pattern Analysis and Machine Intelligence*, 33(5):898–916, August 2011.

[5] Ahsan Javed Awan, Mats Brorsson, Vladimir Vlassov, and Eduard Ayguade. Performance Characterization of In-Memory Data Analytics on a Modern Cloud Server. In *Proceedings of the Fifth IEEE International Conference on Big Data and Cloud Computing*, pages 1–8, Dalian, China, August 2015.

[6] Peter Bajcsy, Antoine Vandecreme, Julien Amelot, Phuong Nguyen, Joe Chalfoun, and Mary Brady. Terabyte-sized Image Computations on Hadoop Cluster Platforms. In *Proceedings of 2013 IEEE International Conference on Big Data*, pages 729–737, Silicon Valley, CA, October 2013.

[7] Peter J Braam. The Lustre Storage Architecture. *Cluster File Systems, Inc*, 2004.

[8] Paul G Brown. Overview of SciDB: Large Scale Array Storage, Processing and Analysis. In *Proceedings of the 2010 ACM SIGMOD International Conference on Management of data*, pages 963–968, Indianapolis, IN, June 2010.

[9] Xiangping Bu, Jia Rao, and Cheng-zhong Xu. Interference and Locality-aware Task Scheduling for MapReduce Applications in Virtual Clusters. In *Proceedings of the 22nd International Symposium on High-performance Parallel and Distributed Computing*, pages 227–238, New York, NY, June 2013.

[10] Nicholas Chaimov, Allen Malony, Shane Canon, Costin Iancu, Khaled Z Ibrahim, and Jay Srinivasan. Scaling Spark on HPC Systems. In *Proceedings of the 25th ACM International Symposium on High-Performance Parallel and Distributed Computing*, pages 97–110, Kyoto, Japan, May 2016.

[11] Zeqiang Chen, Nengcheng Chen, Chao Yang, and Liping Di. Cloud Computing Enabled Web Processing Service for Earth Observation Data Processing. *IEEE Journal of Selected Topics in Applied Earth Observations and Remote Sensing*, 5(6):1637–1649, July 2012.

[12] Dazhao Cheng, Jia Rao, Changjun Jiang, and Xiaobo Zhou. Resource and Deadline-aware Job Scheduling in Dynamic Hadoop Clusters. In *Proceedings of the 2015 IEEE International Parallel and Distributed Processing Symposium*, pages 956–965, Hyderabad, India, July 2015.

[13] Petcu Dana, Panica Silviu, Neagul Marian, Frîncu Marc, Zaharie Daniela, Ciorba Radu, and Dinis Adrian. Earth Observation Data Processing in Distributed Systems. *Informatica: An International Journal of Computing and Informatics*, 34(4):463–476, December 2010.

[14] Jeffrey Dean and Sanjay Ghemawat. MapReduce: Simplified Data Processing on Large Clusters. *Communications of the ACM*, 51(1):107–113, January 2008.

[15] Christina Delimitrou and Christos Kozyrakis. Quasar: Resource-efficient and QoS-aware Cluster Management. In *Proceedings of the 19th International Conference on Architectural Support for Programming Languages and Operating Systems*, pages 127–144, Salt Lake City, UT, March 2014.

[16] Ahmed Eldawy and Mohamed F Mokbel. A Demonstration of Spatialhadoop: An Efficient MapReduce Framework for Spatial Data. *Proceedings of the VLDB Endowment*, 6(12):1230–1233, August 2013.

[17] Ali Ghodsi, Matei Zaharia, Benjamin Hindman, Andy Konwinski, Scott Shenker, and Ion Stoica. Dominant Resource Fairness: Fair Allocation of Multiple Resource Types. In *Proceedings of the 8th USENIX Conference on Networked Systems Design and Implementation*, pages 323–336, Boston, MA, March 2011.

[18] Roberto Giachetta. A Framework for Processing Large Scale Geospatial and Remote Sensing Data in MapReduce Environment. *Computers & Graphics*, 49(C):37–46, June 2015.

[19] Google Compute Engine. `https://cloud.google.com/compute/`. Accessed: 6 March, 2018.

[20] Lei Gu and Huan Li. Memory or Time: Performance Evaluation for Iterative Operation on Hadoop and Spark. In *Proceedings of the 10th IEEE International Conference on High Performance Computing and Communications & IEEE International Conference on Embedded and Ubiquitous Computing*, pages 721–727, Zhangjiajie, China, November 2013.

[21] Gavin Hackeling. *Mastering Machine Learning With Scikit-learn*. Packt Publishing, 2014.

[22] Hadoop: Capacity Scheduler. `https://hadoop.apache.org/docs/stable/hadoop-yarn/hadoop-yarn-site/CapacityScheduler.html`. Accessed: 19 June, 2017.

[23] Hadoop: Fair Scheduler. `https://hadoop.apache.org/docs/stable/hadoop-yarn/hadoop-yarn-site/FairScheduler.html`. Accessed: 19 June, 2017.

[24] Herodotos Herodotou, Harold Lim, Gang Luo, Nedyalko Borisov, Liang Dong, Fatma Bilgen Cetin, and Shivnath Babu. Starfish: A Self-tuning System for Big Data Analytics. In *Proceedings of the Conference on Innovative Data Systems Research*, volume 11, pages 261–272, 2011.

[25] Benjamin Hindman, Andy Konwinski, Matei Zaharia, Ali Ghodsi, Anthony D Joseph, Randy H Katz, Scott Shenker, and Ion Stoica. Mesos: A Platform for Fine-grained Resource Sharing in the Data Center. In *Proceedings of the 8th USENIX Conference on Networked Systems Design and Implementation*, pages 295–308, Boston, MA, March 2011.

[26] Shengsheng Huang, Jie Huang, Jinquan Dai, Tao Xie, and Bo Huang. The HiBench Benchmark Suite: Characterization of the MapReduce-based Data Analysis. In *Proceedings of the 26th IEEE International Conference on Data Engineering Workshops (ICDEW)*, pages 41–51, Long Beach, CA, March 2010.

[27] Wei Huang, Lingkui Meng, Dongying Zhang, and Wen Zhang. In-Memory Parallel Processing of Massive Remotely Sensed Data Using an Apache Spark on Hadoop YARN Model. *IEEE Journal of Selected Topics in Applied Earth Observations and Remote Sensing*, 10(1):3–19, January 2017.

[28] Patrick Hunt, Mahadev Konar, Flavio Paiva Junqueira, and Benjamin Reed. ZooKeeper: Wait-free Coordination for Internet-scale Systems. In *Proceedings of the 2010 USENIX Conference on USENIX Annual Technical Conference*, Boston, MA, June 2010.

[29] Roger Ignazio. *Mesos in Action*. Manning Publications Co., 2016.

[30] Michael Isard, Vijayan Prabhakaran, Jon Currey, Udi Wieder, Kunal Talwar, and Andrew Goldberg. Quincy: Fair Scheduling for Distributed Computing Clusters. In *Proceedings of the 22nd ACM Symposium on Operating Systems Principles*, pages 261–276, Big Sky, MT, October 2009.

[31] Dharmesh Kakadia. *Apache Mesos Essentials*. Packt Publishing Ltd, 2015.

[32] Holden Karau, Andy Konwinski, Patrick Wendell, and Matei Zaharia. *Learning Spark: Lightning-fast Big Data Analysis*. O'Reilly Media, Inc., 2015.

[33] Kamal Kc and Kemafor Anyanwu. Scheduling Hadoop Jobs to Meet Deadlines. In *Proceedings of the 2nd IEEE International Conference on Cloud Computing Technology and Science*, pages 388–392, Indianapolis, IN, November 2010.

[34] Min Li, Jian Tan, Yandong Wang, Li Zhang, and Valentina Salapura. SparkBench: A Comprehensive Benchmarking Suite for in Memory Data Analytic Platform Spark. In *Proceedings of the 12th ACM International Conference on Computing Frontiers*, pages 53:1–53:8, Ischia, Italy, May 2015.

[35] Yunpeng Li, David J Crandall, and Daniel P Huttenlocher. Landmark Classification in Large-scale Image Collections. In *Proceedings of the 12th IEEE International Conference on Computer Vision*, pages 1957–1964, Kyoto, Japan, September 2009.

[36] Fan Liang, Chen Feng, Xiaoyi Lu, and Zhiwei Xu. Performance Benefits of DataMPI: A Case Study with BigDataBench. In *Proceedings of the Workshop on Big Data Benchmarks, Performance Optimization, and Emerging Hardware*, pages 111–123, Salt Lake City, UT, March 2014.

[37] Feng-Cheng Lin, Lan-Kun Chung, Chun-Ju Wang, Wen-Yuan Ku, and Tien-Yin Chou. Storage and Processing of Massive Remote Sensing Images Using a Novel Cloud Computing Platform. *GIScience & Remote Sensing*, 50(3):322–336, July 2013.

[38] Jia-Chun Lin and Ming-Chang Lee. Performance Evaluation of Job Schedulers on Hadoop YARN. *Concurrency and Computation: Practice and Experience*, 28(9):2711–2728, February 2016.

[39] David G Lowe. Distinctive Image Features From Scale-invariant Keypoints. *International Journal of Computer Vision*, 60(2):91–110, November 2004.

[40] Andre Luckow, Ioannis Paraskevakos, George Chantzialexiou, and Shantenu Jha. Hadoop on HPC: Integrating Hadoop and Pilot-based Dynamic Resource Management. In *Proceedings of the 2016 IEEE International Parallel and Distributed Processing Symposium Workshops*, pages 1607–1616, Chicago, IL, August 2016.

[41] Yan Ma, Haiping Wu, Lizhe Wang, Bormin Huang, Rajiv Ranjan, Albert Zomaya, and Wei Jie. Remote Sensing Big Data Computing: Challenges and Opportunities. *Future Generation Computer Systems*, 51:47–60, October 2015.

[42] Matt Massie, Bernard Li, Brad Nicholes, Vladimir Vuksan, Robert Alexander, Jeff Buchbinder, Frederiko Costa, Alex Dean, Dave Josephsen, Peter Phaal, and Daniel Pocock. *Monitoring with Ganglia: Tracking Dynamic Host and Application Metrics at Scale*. O'Reilly Media, Inc., 2012.

[43] Microsoft Azure. `https://azure.microsoft.com/`. Accessed: 6 March, 2018.

[44] Arun C Murthy, Vinod Kumar Vavilapalli, Doug Eadline, Joseph Niemiec, and Jeffrey Markham. *Apache Hadoop YARN: Moving Beyond MapReduce and Batch Processing with Apache Hadoop 2*. Pearson Education, 2014.

[45] Amit Nandi. *Spark for Python Developers*. Packt Publishing, 2015.

[46] Christopher Olston, Benjamin Reed, Utkarsh Srivastava, Ravi Kumar, and Andrew Tomkins. Pig Latin: A Not-so-foreign Language for Data Processing. In *Proceedings of the 2008 ACM SIGMOD International Conference on Management of Data*, pages 1099–1110, Vancouver, Canada, June 2008.

[47] Kay Ousterhout, Ryan Rasti, Sylvia Ratnasamy, Scott Shenker, and Byung-Gon Chun. Making Sense of Performance in Data Analytics Frameworks. In *Proceedings of the 12th USENIX Conference on Networked Systems Design and Implementation*, volume 15, pages 293–307, Oakland, CA, May 2015.

[48] Antonio Plaza, David Valencia, Javier Plaza, and Pablo Martinez. Commodity Cluster-based Parallel Processing of Hyperspectral Imagery. *Journal of Parallel and Distributed Computing*, 66(3):345–358, March 2006.

[49] Antonio J Plaza. Special Issue on Architectures and Techniques for Real-time Processing of Remotely Sensed Images. *Journal of Real-Time Image Processing*, 4(3):191–193, June 2009.

[50] Malte Schwarzkopf, Andy Konwinski, Michael Abd-El-Malek, and John Wilkes. Omega: Flexible, Scalable Schedulers for Large Compute Clusters. In *Proceedings of the 8th ACM European Conference on Computer Systems*, pages 351–364, Prague, Czech Republic, April 2013.

[51] Konstantin Shvachko, Hairong Kuang, Sanjay Radia, and Robert Chansler. The Hadoop Distributed File System. In *Proceedings of the 2010 IEEE 26th Symposium on Mass Storage Systems and Technologies (MSST)*, pages 1–10, Incline Village, NV, May 2010.

[52] Zhongyi Sun, Fengke Chen, Mingmin Chi, and Yangyong Zhu. A Spark-based Big Data Platform for Massive Remote Sensing Data Processing. In *Proceedings of the 2nd International Conference on Data Science*, pages 120–126, Sydney, Australia, August 2015.

[53] Chris Sweeney, Liu Liu, Sean Arietta, and Jason Lawrence. HIPI: A Hadoop Image Processing Interface for Image-based MapReduce Tasks. B.S. Thesis, University of Virginia, 2011.

[54] Jian Tan, Alicia Chin, Zane Zhenhua Hu, Yonggang Hu, Shicong Meng, Xiaoqiao Meng, and Li Zhang. DynMR: Dynamic MapReduce with Reducetask Interleaving and Maptask Backfilling. In *Proceedings of the 9th European Conference on Computer Systems*, pages 2:1–2:14, Amsterdam, The Netherlands, April 2014.

[55] Ashish Thusoo, Joydeep Sen Sarma, Namit Jain, Zheng Shao, Prasad Chakka, Suresh Anthony, Hao Liu, Pete Wyckoff, and Raghotham Murthy. Hive: A Warehousing Solution Over a Map-Reduce Framework. *Proceedings of the VLDB Endowment*, 2(2):1626–1629, August 2009.

[56] Vinod Kumar Vavilapalli, Arun C Murthy, Chris Douglas, Sharad Agarwal, Mahadev Konar, Robert Evans, Thomas Graves, Jason Lowe, Hitesh Shah, Siddharth Seth, et al. Apache Hadoop YARN: Yet Another Resource Negotiator. In *Proceedings of the 4th Annual Symposium on Cloud Computing*, pages 1–16, Santa Clara, CA, October 2013.

[57] Sridhar Vemula and Christopher Crick. Hadoop Image Processing Framework. In *Proceedings of 2015 IEEE International Congress on Big Data (BigData Congress)*, pages 506–513, New York, NY, June 2015.

[58] Shivaram Venkataraman, Zongheng Yang, Michael J Franklin, Benjamin Recht, and Ion Stoica. Ernest: Efficient Performance Prediction for Large-Scale Advanced Analytics. In *Proceedings of the 13th USENIX Symposium on Networked Systems Design and Implementation*, pages 363–378, Santa Clara, CA, March 2016.

[59] Abhishek Verma, Ludmila Cherkasova, and Roy H Campbell. ARIA: Automatic Resource Inference and Allocation for MapReduce Environments. In *Proceedings of the 8th ACM International Conference on Autonomic Computing*, pages 235–244, Karlsruhe, Germany, June 2011.

[60] Abhishek Verma, Luis Pedrosa, Madhukar Korupolu, David Oppenheimer, Eric Tune, and John Wilkes. Large-scale Cluster Management at Google with Borg. In *Proceedings of the 10th European Conference on Computer Systems*, pages 18:1–18:17, Bordeaux, France, April 2015.

[61] Pengyao Wang, Jianqin Wang, Ying Chen, and Guangyuan Ni. Rapid Processing of Remote Sensing Images Based on Cloud Computing. *Future Generation Computer Systems*, 29(8):1963–1968, October 2013.

[62] Tom White. *Hadoop: The Definitive Guide*. O'Reilly Media, Inc., 2015.

[63] Jin Xing and Renée Sieber. Sampling Based Image Splitting in Large Scale Distributed Computing of Earth Observation Data. In *Geoscience and Remote Sensing Symposium (IGARSS), 2014 IEEE International*, pages 1409–1412, Quebec City, Canada, July 2014.

[64] Yuzhong Yan and Lei Huang. Large-scale Image Processing Research Cloud. *Cloud Computing*, pages 88–93, May 2014.

[65] Zhengwei Yang, Liping Di, Genong Yu, and Zeqiang Chen. Vegetation Condition Indices for Crop Vegetation Condition Monitoring. In *Proceedings of the 2011 IEEE International Geoscience and Remote Sensing Symposium*, pages 3534–3537, Vancouver, Canada, July 2011.

[66] Yi Yao, Han Gao, Jiayin Wang, Ningfang Mi, and Bo Sheng. OPERA: Opportunistic and Efficient Resource Allocation in Hadoop YARN by Harnessing Idle Resources. In *Proceedings of the 25th International Conference on Computer Communication and Networks*, pages 1–9, Waikoloa, HI, August 2016.

[67] Matei Zaharia, Dhruba Borthakur, Joydeep Sen Sarma, Khaled Elmeleegy, Scott Shenker, and Ion Stoica. Delay Scheduling: A Simple Technique for Achieving Locality and Fairness in Cluster Scheduling. In *Proceedings of the 5th European conference on Computer systems*, pages 265–278, Paris, France, April 2010.

[68] Matei Zaharia, Mosharaf Chowdhury, Tathagata Das, Ankur Dave, Justin Ma, Murphy McCauley, Michael J Franklin, Scott Shenker, and Ion Stoica. Resilient Distributed Datasets: A Fault-Tolerant Abstraction for In-memory Cluster Computing. In *Proceedings of the 9th USENIX conference on Networked Systems Design and Implementation*, pages 15–28, San Jose, CA, April 2012.

[69] Matei Zaharia, Mosharaf Chowdhury, Michael J Franklin, Scott Shenker, and Ion Stoica. Spark: Cluster Computing with Working Sets. *Proceedings of the 2nd USENIX conference on Hot topics in cloud computing*, pages 1–7, June 2010.

[70] Jixian Zhang. Multi-source Remote Sensing Data Fusion: Status and Trends. *International Journal of Image and Data Fusion*, 1(1):5–24, February 2010.

[71] Qi Zhang, Mohamed Faten Zhani, Yuke Yang, Raouf Boutaba, and Bernard Wong. PRISM: Fine-grained Resource-aware Scheduling for MapReduce. *IEEE Transactions on Cloud Computing*, 3(2):182–194, January 2015.

[72] Xuechen Zhang, Ujjwal Khanal, Xinghui Zhao, and Stephen Ficklin. Understanding Software Platforms for In-memory Scientific Data Analysis: A Case Study of the Spark System. In *Proceedings of the 22nd IEEE International Conference on Parallel and Distributed Systems (ICPADS)*, pages 1135–1144, Wuhan, China, December 2016.

[73] Barbara Zitova and Jan Flusser. Image Registration Methods: A Survey. *Image and Vision Computing*, 21(11):977–1000, 2003.