A Case Study of Spark Resource Configuration and Management for Image Processing Applications

Owolabi Adekoya
Department of Computer Science,
University of Saskatchewan
Saskatoon, SK, CANADA
ooa712@mail.usask.ca

Habib Sabiu
Department of Computer Science,
University of Saskatchewan
Saskatoon, SK, CANADA
has956@mail.usask.ca

Derek Eager
Department of Computer Science,
University of Saskatchewan
Saskatoon, SK, CANADA
eager@cs.usask.ca

Winfried Grassmann
Department of Computer Science,
University of Saskatchewan
Saskatoon, SK, CANADA
grassman@cs.usask.ca

Dwight Makaroff
Department of Computer Science,
University of Saskatchewan
Saskatoon, SK, CANADA
makaroff@cs.usask.ca

ABSTRACT

The world population is expected to reach an estimated 9.8 billion by 2050, necessitating substantial increases in food production. Achieving such increases will require large-scale application of computer informatics within the agricultural sector. In particular, application of informatics to crop breeding has the potential to greatly enhance our ability to develop new varieties quickly and economically. Achieving this potential, however, will require capabilities for analyzing huge volumes of data acquired from various field-deployed image acquisition technologies. Although numerous frameworks for big data processing have been developed, there are relatively few published case studies that describe user experiences with these frameworks in particular application science domains.

In this paper, we describe our efforts to apply Apache Spark to three applications of initial interest within the Plant Phenotyping and Imaging Research Centre (P^{2}IRC) at the University of Saskatchewan. We find that default Spark parameter settings do not work well for these applications. We carry out extensive performance experiments to investigate the impact of alternative Spark parameter settings, both for applications run individually and in scenarios with multiple concurrently executing applications. We find that optimizing Spark parameter settings is challenging, but can yield substantial performance improvements, particularly with concurrent applications, provided that the dataset characteristics are considered. This is a first step towards insights regarding Spark parameter tuning on these classes of applications that may be more generally applicable to broader ranges of applications.

CCS CONCEPTS

• **Information systems → Information systems applications**; **Computing platforms**;

KEYWORDS

Spark, Big Data Performance, Image Processing, Machine Learning

ACM Reference Format:
Owolabi Adekoya, Habib Sabiu, Derek Eager, Winfried Grassmann, and Dwight Makaroff. 2018. A Case Study of Spark Resource Configuration and Management for Image Processing Applications. In *Proceedings of CASCON 2018 (CASCON'18).* ACM, New York, NY, USA, 12 pages.

1 INTRODUCTION

The Plant Phenotyping and Imaging Research Centre (P^{2}IRC) was founded in 2015 with $37.2 million awarded to the University of Saskatchewan by the Canada First Research Excellence Fund. P^{2}IRC researchers are exploring various analysis techniques to elicit insight into the way plants grow, and developing prediction and monitoring applications to help breeders select traits for a safer food supply for the world. Tolerance of extreme environmental conditions, pest resistance, and increased yield/nutrition are some of the characteristics that the next generation of crops will require.

A major focus of P^{2}IRC concerns technologies for plant image acquisition and analysis. Collection of basic image data from field test plots began in the summer of 2016 using fixed-position time-lapse cameras as well as drones equipped with high-fidelity multispectral cameras. Although initial data sets are relatively small, as more advanced imaging technologies are deployed the quantity of image data is expected to greatly increase, necessitating the use of distributed data storage and analysis techniques.

The MapReduce model [3] provides a basis for several distributed processing systems. Apache Hadoop[1] was developed as an open source software framework for distributed storage and processing using MapReduce, and has been widely used [14]. Apache Spark [21] was developed in recognition of MapReduce model limitations. Spark extends MapReduce using a data-sharing abstraction called "Resilient Distributed Datasets" (RDDs), through which a variety of types of workloads can be efficiently supported. The potential benefits of Spark and the convenience of its use, for example its Pyspark[2] Python interface, made Spark a possible framework for use within P^{2}IRC.

[1]http://hadoop.apache.org (Accessed: Jul 12, 2018)
[2]http://spark.apache.org/docs/latest/api/python/ (Accessed: Jul 12, 2018)

Performance evaluation of distributed processing frameworks is complicated by the substantial number of tuneable parameters such frameworks typically offer. Our evaluation of Spark therefore entailed use of extensive performance experiments to determine the impact of alternative Spark parameter settings. As our initial datasets were relatively small, we were able to perform these experiments on two small clusters, as well as run experiments on a single well equipped server machine for comparison purposes. We evaluated Spark with three applications of initial interest in P^2IRC: *flowerCounter*, *imageRegistration*, and *imageClustering*. Note that these applications were not selected owing to any particular inherent affinity to the Spark processing model; in particular, only one of the applications (*imageClustering*) produces intermediate data output that can be persisted in memory using RDDs.

Of interest are both application performance when run in isolation, and when multiple applications are run concurrently. In the latter case, to share cluster resources efficiently, the multiple applications submitted to the cluster must be appropriately scheduled and resourced by a cluster manager. Options for the cluster manager in Spark include a) Standalone which comes as part of the Spark distribution, b) Apache Mesos [7], and c) Hadoop YARN [17]. All three of these managers were evaluated in our experiments.

The main contributions of this paper are the following:

(1) We provide a case study investigating the use of Spark in the plant phenotyping/imaging application domain.
(2) We identify the performance impact of the various Spark resource allocation parameters on three domain specific image processing applications.
(3) We perform an in-depth analysis of the performance implication of the two Spark resource allocation modes (static and dynamic) and the three Spark supported cluster managers, on concurrent multiple instances of the applications.
(4) We quantify the effect of key Spark configuration parameters on cluster resource utilization and makespan.

The rest of this paper is organized as follows. Section 2 provides an overview of the related work in this area, while Section 3 expands on our motivations. Our experimental design is described in Section 4. We discuss the results and implications of our experiments in Section 5. Finally, Section 6 summarizes our contributions and outlines some areas for future work.

2 BACKGROUND AND RELATED WORK

This section describes the components of Spark applications and the related research work that evaluates the performance of Spark in various applicable deployments. For the purposes of this paper, we restrict ourselves to research regarding resource allocation and cluster management.

Big Data Frameworks such as Spark run *tasks* of *jobs/applications* in parallel on cluster compute nodes. A Spark application consists of five major components, briefly discussed as follows:

- Driver Program - an application written using any of the supported Spark APIs - Scala, Python, Java or R.
- Cluster Manager - the management software Spark uses to acquire worker resources for executing the driver program.
- Worker - provides compute resources such as the CPU, memory and storage. Each worker manages at least one *Executor-Backend* process to launch and manage executor instances.
- Executor - a Java Virtual Machine (JVM) process created by Spark on each worker node for an application. Each executor maintains a thread pool for tasks.
- Task - This is the smallest unit of work sent by the *TaskScheduler* to an executor. Tasks operate on RDDs and perform transformation/data transfer operations.

Tasks may be part of a single application run exclusively on a cluster, or there may be many simultaneous application shared between multiple users/organizations. These factors require mechanisms and policies for allocating cluster resources and scheduling tasks/jobs/applications on the available compute nodes.

The default Spark Standalone cluster manager schedules applications in FIFO order, and each application will try to use all available cluster resource by default. Subsequent applications submitted will be queued and must wait for the first application to run to completion. Application developers may set *spark.cores.max* to limit the number of cores an application requests and thus permit multiple concurrent users or use dynamic allocation as explained later.

Apache Mesos [7] is a 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 used between multiple frameworks as if they belonged to a single, large server. Mesos allows organizations 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) [6] to allocate resources to frameworks.

Hadoop YARN [17] allows multiple data processing engines with diverse programming models to run alongside Hadoop MapReduce on the same cluster and access centralized datasets stored on the Hadoop Distributed FileSystem (HDFS) [15]. YARN permits deployment of different schedulers and resource allocation policies.

There are two modes of resource allocation. By default, Spark uses *Static Resource Allocation*, in which applications hold on to resources until completion. *Dynamic Resource Allocation* adjusts resource allocation at runtime, based on the workload requirements/available resources. If an executor is idle for more than a threshold of time (60 seconds by default), it is reallocated to a different application. This is useful in a multi-tenant shared cluster where multiple applications need to share cluster resources.

Most existing MapReduce schedulers allocate a fixed set of resources (e.g. CPU and memory) to jobs at the task-level, thus assuming the run-time resource consumption of tasks are stable over its lifetime. Zhang *et al.* [22] argue that this does not provide optimal performance since different tasks can have varying resource requirements over time. 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 performance and resource utilization by allocating resources to phases within an application according to their resource requirements.

The effect of data partition size and executor core scaling to application completion time in data analytics frameworks has been previously investigated [16, 23]. However, the characteristics of the applications considered in those works are different from the applications studied here. The applications used in those studies are generic benchmarks. The applications used in this study are real-world applications that have varying uses of Spark primitives and RDDs. This makes our performance study of particular interest, compared to previous work.

The impact of Spark's use of memory, caching, serialization, local file systems and SSDs is compared to SciDB by Zhang *et al.* [23] for in-memory scientific data analysis. Generational garbage collection, multi-threading and executor scaling influence on TPC-H queries using Spark have also been studied for their performance effects [2].

Straggler tasks in heterogeneous environments have also been studied and an improved version of Spark's speculative execution algorithm has been proposed [20], as existing speculative execution was shown to be ineffective since straggler tasks due to slow nodes are not accurately identified. HAT [1] is an optimized MapReduce Scheduler that mitigates the impact of slow tasks in heterogeneous environments using historical information to detect slow tasks/nodes and then avoids these slow nodes for backup tasks.

Veiga *et al.* [18] compared the performance of Hadoop, Spark and Flink[3] using benchmarks including PageRank [5] and K-Means clustering and revealed that Spark outperformed Hadoop and Flink with respect to scalability across all the benchmarks considered, especially for K-Means. The HDFS partition size most suited for Spark workloads was 64 MB. The authors found that one big executor with 8 cores (the machine maximum) was the best configuration for Spark workloads except for PageRank.

The finding that one big executor per node is the optimal configuration for Spark agrees with Li *et al.* [8]. However, Chiba *et al.* [2] suggests that two or four executors per node achieves better performance for their test workloads.

The influence of configuration settings on application performance was studied by Nguyen *et al.* [11] with different application workloads. The authors developed a framework capable of identifying key configuration parameters that affect performance. Two Spark parameters investigated pertinent to our study are *spark.executor.cores* and *spark.speculation*. For all workloads and executor sizes considered, task durations were greatly improved but speculative execution either negatively impacted task execution or had no effect at all.

Ousterhout *et al.* [13] investigated the presumed major bottlenecks in data analytics frameworks: network, the disk and stragglers. They developed a Blocked Time Analysis methodology that measures how long jobs spent blocked on cluster resources, and applied this methodology for two benchmarks and industry workloads running on Spark. The authors found that that network improvements and disk I/O did not impact performance to a noticeable degree, as CPU was typically the bottleneck. However, results for other environments and workloads may differ.

Different schedulers and cluster managers have been proposed, but are not widely deployed. Cluster managers such as Borg [19] effectively run applications across thousands of machines by grouping jobs into categories based on priority. Quasar [4] designs resource efficient and QoS-aware cluster management which aims to improve cluster resource utilization by eliminating resource reservation. Our workload applications are long running and batch; they do not require strict deadlines, therefore, do not reserve resources.

More study is required for a comprehensive understanding of the influence of configuration parameters on Spark workloads. Conflicting results are reported that are insufficient to be generally applicable. Previous work has helped identify influential configuration parameters. Prior performance studies of Spark are crucial as they provided useful insights about the performance of analytics applications. Our aim is to see whether the widely-supported schedulers and resource managers were vulnerable to previously identified problems with our workload before choosing more experimental components with which to experiment.

3 CONTEXT & MOTIVATION

The P^{2}IRC project has collected still-camera images from test plots in 2016 and 2017. In 2016, up to 10 cameras in separate plots were used to take an image every minute. This generated about 600 GB of data. In 2017, 48 cameras were deployed and images were taken every 5 minutes, but the images were of higher resolution, and 2 TB of data was generated. We have approximately 1 million drone images collected from the 2016 and 2017 growing season for a total of 8 TB of data. This data was collected from 2 small test farms. This level of data storage can be contained in a single server computer, but the deployments increase each year, and the corresponding storage/compute capabilities will not be able to keep pace. Individual applications may take between a day's worth of images from a single camera data from multiple seasons and multiple cameras for analysis, requiring that all the data be available simultaneously from an application's point of view. This suggests a distributed filesystem approach.

The applications used in our study are selected not because of their inherent affinity to the Spark processing model. Spark favours applications that produce intermediate data output which can be persisted in memory for subsequent processing in the pipeline (only *imageClustering* among our benchmarks).

Furthermore, our chosen applications have differing resource requirements and operational phases over their lifetimes. We wished to see if particular types of applications can more efficiently share resources when scheduled simultaneously, or if common optimizations for the influential configuration parameter is reasonable with this set of applications. If the required parameter settings are substantially different, then cluster managers may need to be used to treat similar classes of applications as multiple instances of Spark frameworks that share the same hardware infrastructure.

We intend to determine if the processing needs of P^{2}IRC could be met by Spark cohesively by utilizing HDFS storage and moderate-sized clusters for a subset of the data already collected. If so, this would enable a variety of subsequent applications, potentially specifically suited to the advantages of Spark, Flink or other frameworks to be developed/hosted in this infrastructure. This study provides a first step towards this understanding of the nature of resource utilization/configuration and a methodology for obtaining detailed performance measurements. With the insight gained, we hope to determine favourable deployment scenarios and configurations, and if parameter settings generalizations are possible.

[3]flink.apache.org. Last viewed Aug. 21, 2018

4 EXPERIMENTAL DESIGN

4.1 Datasets

The datasets are still-camera JPEG images and drone PNG images from the summer of 2016. Drone images were processed by *imageRegistration* while *imageClustering* and *flowerCounter* used still camera images.

Drone images. The camera captures images in five bands: Blue, Green, Red, Red Edge and Near-Infrared. Three drone images datasets were used. The first 200 image sets from each dataset were used by *imageRegistration*. Each dataset was 1.7 GB.

Still images for single application experiments. We selected an entire collection period, which could be up to 15 days of the still-camera images from 6 cameras for each dataset. The July dataset is about 35 GB and contains images collected from July 1-15, 2016. We used images from August 26-31 (about 20 GB), and images collected from September 9-12 (about 6 GB).

Still images for multiple application experiments. For *flowerCounter*, three single-day datasets are used. The first 2 datasets have 1020 images each while the third has 719 images, because the camera was not set up at the beginning of the day. The datasets are 314 MB, 354, and 422 MB in size. For *imageClustering*, each dataset contained between 3 and 4 day's worth of images, for dataset sizes of 1.3 GB, 1.4 GB, and 0.85 GB, respectively.

Individual image files are relatively small (500 KB) compared to the default HDFS block size of 128 MB. For experimental purposes, this had little effect on performance, as the Spark *repartition* operation and *spark.files.maxPartitionBytes* parameter were used to pack images into a single RDD partition.

As well, none of the datasets can be truly considered Big Data. Each dataset could easily be stored on a single disk and processed for a single application execution on even these small datasets ranges between 30 minutes to 2 hours on one of the clusters using default settings.

4.2 Benchmark Applications

The sequential applications were written in Python, using OpenCV and OpenCV-contrib for various image processing algorithms. They all use *numpy* arrays to store the component values of each channel contributing to the image (Red, Green, Blue, Infrared and Near Infrared), as appropriate. Standard image processing techniques are employed, such as Scale Invariant Feature Transform (SIFT) and K-Means clustering, though not necessarily the same implementations of these functions. These applications are relatively representative in that they cover a diverse set of Spark transformations and actions including *map*, *filter*, *aggregate*, *coalesce*, *sortBy* and *reduceByKey*. Table 1 summarizes the main features of the applications. The Spark transformations have narrow dependencies (e.g. map, filter) where each parent RDD is needed by at most one child RDD as well as wide dependencies (e.g. reduceByKey) where the parent RDD is needed by multiple child RDDs.

imageRegistration. To register multiple images on the same coordinate system, one channel is selected as the reference image and the others are the target images. The registration involves spatially registering the target image(s) to align with the reference image. The sequential version reads the input images and registers multiple channels of images by using OpenCV's implementation of the SIFT algorithm [9] to extract image features. These features are passed to a feature-matching function which takes the descriptor of one feature in the reference image and matches it with all features in the target images using a distance metric. For each target image, the feature with the closest distance is returned. This embarrassingly parallel application aligns a group of 5 input images together on a single coordinate system. The outputs are a set of 5 registered images, a coloured RGB image, a cropped version of the RGB, and an NDVI (Normalized Difference Vegetation Index) image.

flowerCounter. The input images are first passed to a *K-Means* clustering stage. This stage groups all the input images into clusters, based on the percentage of yellow pixels. The selected images from the appropriate cluster are sorted based on the percentage of yellow pixels and passed to a pipeline of various image processing techniques to estimate the number of flowers. The processing pipeline includes converting the image colourspace, computing sigmoid mapping, and finally detecting blobs. Each blob is considered a flower. Final desired output is a text file containing image names and estimated number of flowers per image. The stages in the Flower Counter are of two types: 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 (e.g. *K-Means* clustering).

imageClustering. This application clusters images based on features using the *K-Means* clustering algorithm. The input images are converted to Spark RDDs of numpy arrays and passed to a map function that uses SIFT to extract features and compute descriptors from the input images sequentially within each partition. The extracted feature descriptors are filtered and intermediate output RDDs are collected, and the resulting data structure is passed to the *K-Means* clustering function for model building, training and clustering. The clustering results are reduced into one RDD partition and written to HDFS as a text file containing the names of the images and the associated cluster. This application is CPU-bound during K-means iteration and I/O-bound during clustering.

4.3 Experimental Environment

All of the computers in our experiments ran Ubuntu 16.04 LTS. The baseline system was a single machine configured with 2 Intel(R) Xeon(R) E5-2690 v4 CPUs (14 cores @ 2.60GHz, hyper-threaded with 56 virtual cores), 620 GB RAM and 43 TB of disk space. The clusters were configured as follows:

- Cluster 1: 12 machines, configured as 11 workers and 1 master, running KVM Virtual Machines, connected via a private 1GB Ethernet network. The master and 9 of the workers have Intel(R) Core(TM) i7-2600 CPUs (4 cores @ 3.40GHz, hyper-threaded with 8 virtual cores), and 16 GB of RAM each, while 2 of the workers have Intel(R) Xeon(R) E5-2403 CPUs (4 cores @ 1.80GHz hyper-threaded with 8 virtual cores), and 40 GB of RAM. 1 GB of RAM and 1 virtual core were reserved for the OS.
- Cluster 2: 3 machines, 2 workers and 1 master, running KVM Virtual Machines as well, connected via the campus network with 1 GB ethernet interfaces. The master has 2 Intel(R) Xeon(R) E5-2680 v4 CPUs (2x14 cores @ 2.40GHz, hyper-threaded for 56 virtual cores) and 256 GB of RAM, while one

Table 1: Application Characteristics

Features	**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
Machine Learning Details	-	-	scikit k-means	MLlib k-means	scikit k-means	MLlib k-means
Main Image Processing	Features extraction/matching, perspective transformations		Clustering, color space conversion, sigmoid mapping, blob detection		Features extraction, image clustering	
Spark transformations/actions	map, reduceByKey, foreach		aggregate, map, filter, sortBy, coalesce		map, filter, flatMap, coalesce	

of the workers has 2 Intel(R) Xeon(R) E5-2690 v3 CPUs (12 cores @ 2.60GHz, hyper-threaded for 48 virtual cores) with 384 GB of RAM and the other has 2 Intel(R) Xeon(R) E5-2680 v4 CPUs (2x14 cores @ 2.40GHz, hyper-threaded for 56 virtual cores) with 384 GB of RAM. To keep the configurations comparable, a small amount of RAM was reserved for the host OS and 48 cores were given to each VM, since that is the number of cores available on the smallest host machine (no core reserved for host OS).

We used Hadoop/YARN 2.7.2 and Spark 2.1.0, and Mesos 1.2.0, with Python 2.7.12. JRE 1.8 was the Java version deployed.

4.4 Measurement Methodology

The main metric that we are concerned with is makespan, the time between the submission of the first of a set of applications and the completion of the last application. We also measure a number of other quantities that help us understand how makespan is influenced. These include average CPU and memory utilization, job waiting and execution times, stage level task execution times, number of tasks per node, network/disk throughput, and HDFS I/O.

The Spark Web UI provides various information about applications at run-time. This data is logged to disk for later analysis. The single application experiments used an instrumented version of Spark called SparkOscope[4] to collect CPU, memory, network and disk utilization information in the cluster, as well as some publicly available scripts for visualizing and analyzing Spark logs [12, 23]. The multiple application experiments recorded the beginning and the end time of each experimental run. The external monitoring system Ganglia [10] (version 3.6.0) was used to get cluster CPU and memory utilization of running applications.

4.5 Parameters & Experimental Settings

The results section describes the performance of a single run per configuration unless specified otherwise. The time required to complete a single run was on the order of hours, thus many replications are impractical. For validation purposes, some runs were done multiple times and the variations between runs was minimal, except for the first run after the cluster was rebooted. The quantitative effect of this variation is described at the beginning of Section 5 and therefore that first run is ignored.

[4]https://github.com/ibm-research-ireland (Accessed: 23 Apr, 2018)

We evaluated the effect of a number of Spark configuration parameters, including executor size (CPU cores and memory), RDD partition size, dynamic resource allocation, and speculative execution. All other configuration parameters were left at default settings.

Single application experiments. The sequential version of *flowerCounter* on the baseline system (56 total cores) was compared to the corresponding parallel versions. The closest equivalent of compute cores on Cluster 1 is 9 worker nodes with 6 cores per node, (54 total cores) and on Cluster 2 is 32 cores per node (64 total cores)

As the right partition size is important to avoid data skew and reduce computation time, *flowerCounter* experiments were performed with Executor Sizes between 2 MB and 128 MB in powers of 2. A set of experiments were conducted with all the datasets by increasing the number of cores allocated to each Spark executor, varying the number of executors, keeping the heap size constant based on the total memory allocated to the worker machine. This evaluates the performance impact of the number and size of Spark executors.

Next, Spark's caching mechanism was investigated. The experiments were performed with both *flowerCounter* and *imageClustering* on Cluster 2. The execution time with 10 GB memory and 16 cores per executor was the baseline. Different storage levels in Spark were studied for both applications on Cluster 1 (9 worker nodes) using the July dataset for *flowerCounter*, and the September dataset for *imageClustering*: memory only, disk only and both.

The scale-out design was evaluated by increasing the compute nodes from 1 to 11 (with odd number increments) on Cluster 1 for *flowerCounter* using the July dataset with 128 MB partitions. Single node performance was used as the basis for calculating speedup. Speculative execution experiments were performed with all the nodes in Cluster 1. The experiments used all the datasets for *flowerCounter*, but *imageClustering* used only the July dataset.

Multiple application experiments. We then executed the distributed versions of the benchmark applications on the three cluster managers on Cluster 1 with 11 workers/77 cores. All the experiments only examined one factor at a time (except for parameters that control Spark executor size), instead of multiple factors simultaneously.

This is reasonable, since it was prohibitively time consuming to test the combination of all factors.

Each experimental run involves submitting 3 instances of each of the 3 applications, with five minutes delay between submissions. The 9 application instances were randomly ordered, with the same seed to ensure the same submission sequence. The various executor sizes were as follows: a) 1 core/2 GB RAM, b) 3 core/6 GB RAM, and c) 6 core/12 GB RAM. Each of the executor configurations was accompanied by 3 RDD partition sizes (128 MB, 64 MB and 32 MB): a total of 9 configurations. In addition, each configuration was tested with and without dynamic resource allocation enabled on the 3 Spark-supported cluster managers.

Replication is not performed for every configuration, since the makespan on a few initial replications showed consistent results. A small number of replications on Cluster 1 running without KVM were run with twice the amount of data and/or twice the number of instances to asses at what point variation became a factor.

5 RESULTS

It should be noted that the sequential version in python is restricted to running as a single thread per Linux process, since only one thread can be active at any point in time, due to the python Global Interpreter Lock (GIL).[5] The parallel version uses at least as many cores as available executors. When OpenCV functions are called, the GIL is released, and C/C++ modules can run in multiple threads.

The applications have differing amounts of parallel code in their execution path. While running the sequential applications on the single server, we sampled the CPU utilization. For *flowerCounter*, occasional usage of 8-10 cores simultaneously was observed. On the other hand, a large portion of the execution time for *imageClustering* showed over 50 cores being used at 100%. Evaluating a parallel version on a single machine Spark instantiation could provide an even more interesting performance comparison.

5.1 Validation of Makespan Variation

To investigate performance differences between a freshly started cluster and a cluster that has been running for some time, a set of experiments were done using the initial parameter setup for multiple applications. Five identical runs were done on each cluster manager. Figure 1 shows the makespan results of multiple runs of the same experimental setup on the three cluster managers. The first run deviated from the four subsequent runs in all configurations.

We calculate the mean and standard deviation of the subsequent runs, and used *t-test* with 3 degrees of freedom and $\alpha = 0.05$ to determine if the first run could belong to the remainder of the distribution. The *t-scores* of the first run on Standalone, YARN and Mesos are 29.69, 30.17, and 23.68 respectively. The corresponding *t-value* is 2.353, thus the first experimental run on all the three cluster managers is greater than the *t-value.* The standard deviation of remaining replications is approximately 1% of the mean. Subsequent analysis revealed that the *qemu* process did not have sufficient physical memory on startup which resulted in substantial paging activity during the first run, noticeably slowing its execution.

Furthermore, runs with double the dataset size had twice the number of executors. For *imageClustering*, the OpenCV functions

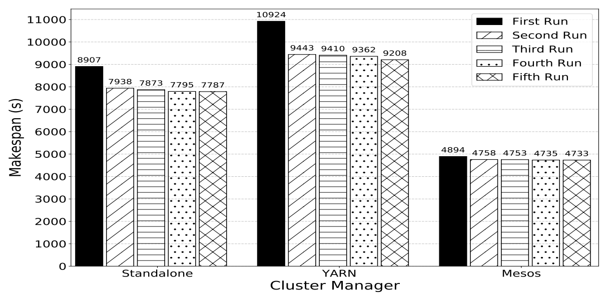


Figure 1: Multiple run variation validation: static/1core-2 GB/32 MB/no spec

with many concurrent threads led to a slowdown of tasks of the other applications on the same host, and substantial increase in task execution time variation. Variation in overall makespan was affected somewhat, but still within 10% of the mean, as the same amount of work must be completed overall. More work is needed to explore the stability of the results.

5.2 Individual Application Performance

Comparison with Sequential Application. Sequential experiments were conducted for only the *flowerCounter* application using all the datasets and compared to the parallel Spark execution on both clusters. The results are shown in Table 2. The speedup obtained on Cluster 1 is approximately 3 times the sequential execution for all datasets with a similar number of compute cores devoted to the work. Similarly, the speedup on Cluster 2 is approximately 4 times the sequential setup for all the datasets.

Table 2: Runtime Comparison: Sequential & Spark *flowerCounter*

Month	Server (56 cores) (minutes)	9-Node (54 cores) (minutes)	9-Node Speedup	2-Node (64 cores) (minutes)	2-Node Speedup
Sep	35.74	11.29	3.2	8.21	4.4
Aug	103.43	28.53	3.6	22.23	4.7
July	231.56	68.45	3.4	54.83	4.2

Effect of Partition Size. The influence of partition size on the total processing time of the *flowerCounter* application is shown in Table 3. These experiments were conducted on Cluster 1 with 9 workers; each executor had 6 cores and 12 GB of RAM. A 64 MB partition size reduces processing time from the 128 MB default. Processing time remained almost the same for further reduction in the partition size until the smallest partition size of 2 MB.

Table 3: Partition Size Comparison:*flowerCounter*

128 MB	64 MB	32 MB	16 MB	8 MB	4 MB	2 MB
4448	3862	3980	3932	4025	4086	4546

[5]https://wiki.python.org/moin/GlobalInterpreterLock (Accessed August 24, 2018)

The stages whose execution times (with respect to the maximum task time) are notably affected by the partition size are the *computeHistogram* and *computeHistogramShifts*. The execution times of the other two stages (*computeFlowerPixelPercentage* and *computeFlowerCount*) are reduced as well but with smaller differences.

The *computeFlowerCount* stage accounted for about 76% and 68% of total completion time, in both instances, respectively. The difference in the execution times for the *computeFlowerCount* stage is very similar, since slow 128 MB partition tasks completed in almost the same time as the larger number of 64 MB partition tasks.

Further investigation into the completion time of each task showed that slow tasks in *other stages* are the cause of the longer completion time for the 128 MB partition; the time for 128 MB is almost twice of 64 MB. It is obvious that with twice as much data, each task should take on average twice as long, but there are half as many tasks, so the elapsed time for hundreds of tasks should be the same. At the stage completion time of about 300 seconds in the *computeHistogram* stage for the 64 MB (with 556 total tasks) configuration, there were still about 39 tasks yet to be completed in the 128 MB (with 279 tasks in total) configuration. These tasks took a disproportionate amount of time to complete, even compared with the previous tasks in that phase.

To further investigate the influence of the partition size on cluster resources, metrics were collected with respect to CPU utilization, memory usage, network and disk throughput. Average values sampled at every 30 seconds were recorded. For CPU usage, there are downward spikes at the transition points between stages. The 128 MB scenario had much longer durations of these spikes, because a few executors were finishing the slow tasks and others were idle, because they could not start the next stage. RAM usage is higher in the first *computeHistogram* stage for 128 MB partitions in comparison to 64 MB partitions. In general, the high CPU and RAM usage by the application in both run scenarios across all the stages of execution in the pipeline is due to large array transformations, involving matrix computation and copying deep array dictionaries. The high CPU volatility especially in the 128 MB run scenario requires further study. Disk and network activity were reasonably similar in each scenario and did not provide further insight.

JVM Executor Scaling. The impact of the number of JVM executors (executor size) on Cluster 1 is investigated next. The number of executors is 18 (3 cores/6 GB RAM each) and 27 (2 cores/4 GB RAM each) for the two configurations (64 MB partition only). The summarized execution time results are shown in Table 4. 27 JVM executors performed better than 18 executors. The differences in execution times are larger in the *computeHistogramShifts* and *computeFlowerPixelPercentage* stages than in the other stages.

For both stages, tasks took more time to complete in the 18 JVM executors instances than in the corresponding 27 JVM executors, especially in the *computeHistogramShifts* stage. In the *computeHistogramShifts* stage, for example, the total task run time in the 18 JVM executors scenario is about twice that of the 27 JVM executors, though the median times were comparable. Some long-running tasks seem to dominate the stage completion time.

For *imageClustering*, the effect of the JVM executor scaling was studied for five iteration steps. Two different experimental scenarios were considered on Cluster 1 with 128 MB partition size (September dataset only): 9 executors (6 cores/12 GB) and 18 executors (3 cores/6 GB). There are 46 tasks in each stage in the pipeline.

Table 4: *flowerCounter*: Task Execution times vs. Executor Size (August Dataset - 19.3 GB)

18 Executor JVMs (64 MB)				
Function Name	Min	Median	Max	Elapsed
computeHistogram	7 s	18 s	35 s	132 s
computeHistogramShift	3	7	36	114
computeFlowerPixelPercentage	2	13	72	138
computeFlowerCount	96	228	408	1440
Total Elapsed Time				1824
27 Executor JVMs (64 MB)				
Function Name	Min	Median	Max	Elapsed
computeHistogram	8	17	31	108
computeHistogramShift	3	7	14	54
computeFlowerPixelPercentage	3	10	21	90
computeFlowerCount	108	222	420	1440
Total Elapsed Time				1692

For the five iteration steps considered, there are eight key stages involved in the application processing whose execution times are dominant. The summarized execution times for the 9 and 18 JVM executors scenarios are shown in Tables 5 and 6. The median execution time for the *collectAsMap* stages is reduced by about 30% for the 18 executor JVMs than for the 9 executor JVMs. Also, the *collectAsMap* stages are more stable in the 18-executor JVM runs. The 9 JVM executors scenario is dominated by slower tasks as reflected in the large variation of the time summary statistics especially in the *collectAsMap* stages and the second *takeSample* stage.

Table 5: Execution Time Summary of 9 JVM Executors for *imageClustering* (128 MB)

Stages	Min	Median	Max	Elapsed
takeSample	498	660	1200	1200
takeSample	0.8	96	162	162
collectAsMap	9	102	192	198
collectAsMap	3	108	174	180
collectAsMap	5	114	174	174
collectAsMap	4	102	168	174
collectAsMap	3	90	210	210
saveAsTextFile	660	780	960	1020
Total Elapsed Time				3318

To further investigate the influence of the number of cores on Cluster 2, more experiments were conducted using all datasets and both applications. The memory size of each executor was kept at 7 GB while *spark.executor.cores* was varied between 1, 4, 8, 12, 16 & 47 cores. For both applications, the single large executor resulted in the slowest processing time. This finding agrees with Chiba *et al.* [2] but contradicts others [8, 18]. The application type has an effect on the best configuration, so this confirms that specific measurements in each environment are necessary [11].

Also, a large number of executor JVMs (94 executors each with one core) degrades performance for both applications; tasks are

Table 6: Execution Time Summary of 18 JVM Executors for *imageClustering* (128 MB)

Stages	Min	Median	Max	Elapsed
takeSample	504	720	780	780
takeSample	47	84	138	138
collectAsMap	48	84	150	150
collectAsMap	52	72	102	102
collectAsMap	50	72	102	102
collectAsMap	50	78	102	102
collectAsMap	50	78	96	102
saveAsTextFile	720	840	1020	1080
Total Elapsed Time				2556

more CPU and/or memory bound (especially with the July dataset). This is due to excessive communication overhead caused by the large number of executors [2].

Impact of Caching. Using different configuration memory settings and *spark.executor.cores* fixed at 16 (2 per node on Cluster 1), experiments were conducted with the July dataset for both applications, but results are shown only for *imageClustering*. Increasing the amount of memory allocated for each executor does not speedup application performance as shown in Figure 2. Some previous studies have shown that allocating more memory for caching RDDs does not always improve performance as workloads sometimes require dynamically distinguishing job stages that are cache friendly and tune accordingly [8].

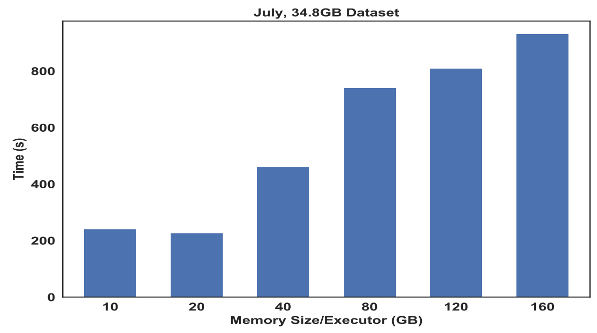


Figure 2: Influence of *spark.executor.memory* - *imageClustering*

The effect of caching the input RDD (input data) using the different storage levels on the applications execution speed was studied. For *flowerCounter*, the input RDD was cached using storage levels including MEMORY_ONLY, DISK_ONLY, MEMORY_AND_DISK, MEMORY_ONLY_SER and MEMORY_AND_DISK_SER while *imageClustering* used DISK_ONLY, MEMORY_AND_DISK and MEMORY_AND_DISK_SER storage levels to cache the input RDD. MEMORY_ONLY failed due to an *out-of-memory* exception.

Data locality policies attempt to assign tasks to nodes near the source data files and instantiate RDDs on the node performing the next task in the pipeline. The locality level summary for both applications is shown in Tables 7 and 8 respectively. For *flowerCounter*, all the storage levels considered showed very similar *runTime* except the MEMORY_AND_DISK_SER storage level whose *runTime* was slightly higher than the other storage levels. With *imageClustering*, MEMORY_AND_DISK storage level exhibited the least *runTime* because of differences in the task locality levels. PROCESS_LOCAL tasks are the most frequent in the *imageClustering* application runs for MEMORY_AND_DISK; the ANY tasks are the least frequent. This contributes to lower execution time.

Table 7: *flowerCounter*: Locality Summary (July, 34.8 GB)

	Task Locality Level Count Summary				
Locality Level	MEM ONLY	DISK ONLY	MEM & DISK	MEM ONLY_SER	MEM & DISK_SER
NODE_LOCAL	225	224	225	225	224
ANY	115	109	104	121	124
PROCESS_LOCAL	778	785	789	772	770

Table 8: *imageClustering*: Locality Summary (Sept., 5.7 GB)

	Task Locality Level Count Summary		
Locality Level	DISK ONLY	MEMORY & DISK	MEMORY & DISK_SER
NODE_LOCAL	547	568	487
ANY	285	145	240
PROCESS_LOCAL	6169	6288	6274

Compute Node Scaling. The node scaling experiments were executed on Cluster 1 with 1 to 11 nodes using all datasets. As seen in Figures 3 and 4, the applications do not exhibit absolute linear scalability but show a quasi-linear trend up to 9 nodes.

The execution time for 11 nodes was greater than that for 9 nodes. This is because 9 worker nodes have twice the CPU frequency of the two remaining nodes. These two workers executed the fewest tasks, but also exhibited slowest task completion time. All other nodes showed similar processing time overall. This behaviour of prolonged tasks is typical of heterogeneous environments and task schedulers fail to handle this dynamically without affecting the overall application performance [1].

Speculative Task Execution. Slow tasks are expected in a heterogeneous cluster environment due to varying node resources. The results obtained from the node scaling experiments clearly validate this premise. Speculation, however, does not automatically make all workloads process faster, depending on the rescheduling behaviour. If the original slow task finished before the rescheduled back-up task, the back-up task would be killed and vice-versa.

Speculation experiments were carried out on both applications using all the datasets to study the effects on straggler tasks. Results from the experiments (3 run instances for each dataset in both scenarios) conducted for *flowerCounter* are shown in Figure 5. Speculation slightly favours *flowerCounter* with all the datasets. Speculation has a large effect for the July and September datasets but has almost no effect with August dataset. On the other hand, for *imageClustering* using the July dataset (not shown), h the processing

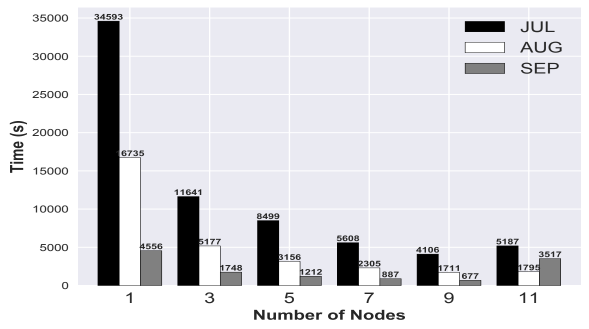


Figure 3: Node Scaling: *Execution Time*

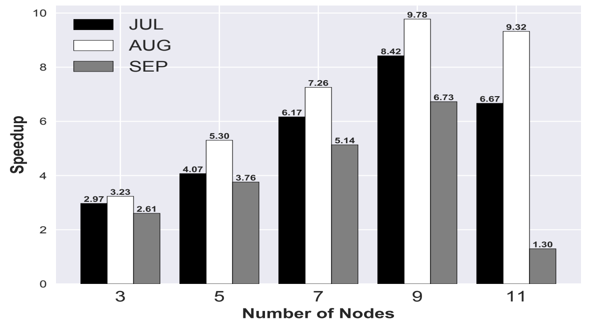


Figure 4: Node Scaling: *Speedup*

time is slightly *increased* with speculation enabled, the difference is miniscule (624 minutes vs. 612 minutes).

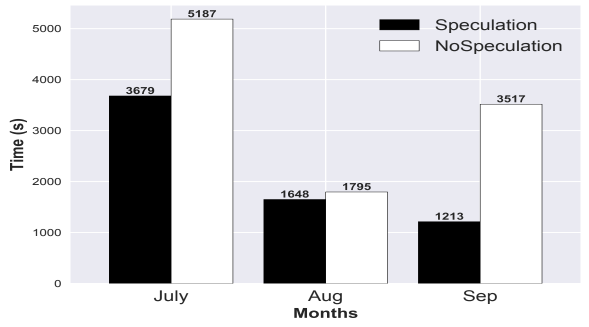


Figure 5: Speculation vs. No speculation (*flowerCounter*)

5.3 Concurrent Application Performance

Impact of Spark Resource Allocation Modes. Figure 6 shows the CPU and memory utilization of the standalone cluster manager. The most CPU-intensive of the applications is *imageRegistration*. The number of tasks in the other 2 apps is much smaller than the cluster core capacity. In particular, when we examine the partitioning behaviour, we see that very few partitions (at most 5) are created for *flowerCounter*. Most cluster resources are idle, so this is clearly a poor operation mode.

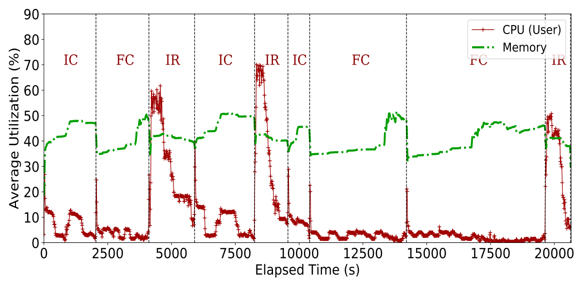


Figure 6: Resource utilization: Standalone/static/1 core-2 GB/128 MB partition/no spec

Figure 7 shows that, with the YARN cluster manager, both CPU and memory utilization gradually increase to approximately 40% as more applications were submitted. This decreases as applications finish execution. After about 6000 seconds, only one application (*imageClustering_job_2*) is left running.

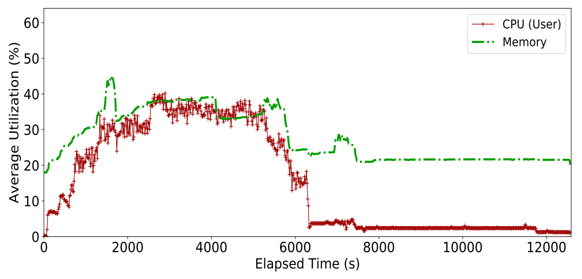


Figure 7: Resource utilization: YARN/static/1 core-2 GB/128 MB partition/no spec

YARN allocates 3 executors to each application by default, which underutilizes cluster resources for all applications. As 9 applications were submitted with this configuration: there are again an excess supply of executors. The cluster administrator must manually set the *spark.executor.instances* configuration parameter to an appropriate value to use all cluster resources. Manually setting this parameter is not easy, since the optimal number of executors depends on the application type and the dataset size.

Figure 8 shows resource utilization with Mesos and static allocation mode. All cluster resources are allocated to the first application and held throughout execution. Upon completion, Mesos reclaims the allocated resources and divides them among all the queued applications. This increases the CPU and memory utilization to approximately 80% and 60%, respectively. When there are resources available, Mesos uses the Dominant Resource Fairness algorithm to fairly share the resources among all waiting applications. The cluster utilization/makespan is dependent on the completion time of the first application and the number of waiting applications at that time. If few resources are released to many waiting applications, little parallelism is realized. Again, one application prolongs the makespan (*flowerCounter_1*).

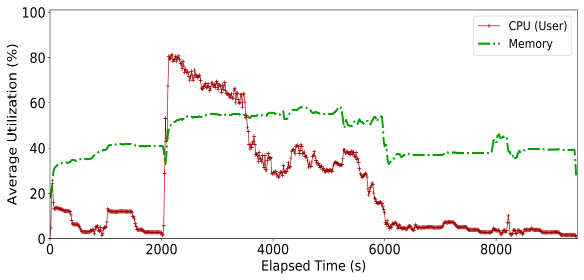


Figure 8: Resource utilization: Mesos/static/1 core-2 GB/128 MB partition/no spec

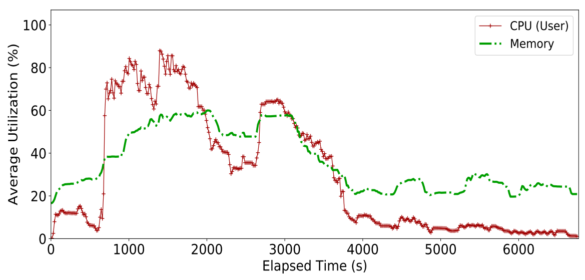


Figure 10: Resource utilization: Mesos/dynamic/1 core-2 GB/128 MB/no spec

The same experiments were repeated using dynamic resource allocation. All other dynamic resource allocation parameters were set to their default values. Figures 9 and 10 present the CPU/memory utilization of Standalone and Mesos. Cluster resource utilization rises to approximately 85% for CPU and 60% for memory on Standalone. Dynamic resource allocation is capable of scaling resource utilization up or down depending on the number of currently submitted tasks. YARN results are similar; the cluster is still substantially under-utilized, due to applications that have few partitions.

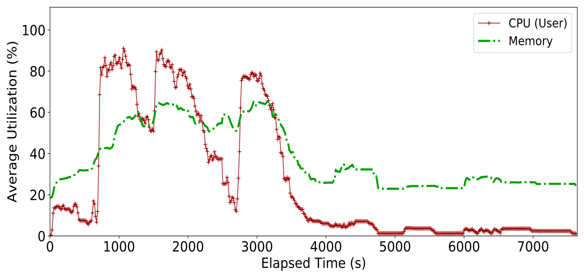


Figure 9: Resource utilization: Standalone/dynamic/1 core-2 GB/128 MB/no spec

Resource Allocation Modes/Partition Size. Table 9 compares the makespan for static resource allocation and different partition sizes. The makespan value decreases when the Spark RDD partition size was decreased due to increased parallelism. This contradicts our earlier experiments where applications are run individually, since the datasets utilized are so small that there are too few tasks available to be scheduled concurrently, and therefore, idle cores.

Table 9: Makespan: Static/1-core/2 GB/no spec

	128 MB	64 MB	32 MB
Standalone	20636	16084	8170
YARN	12562	12007	9826
Mesos	9428	6133	4753

Standalone provides the worst makespan for 128 MB and 64 MB RDD partitions. Although YARN allocated only 3 containers per application (1 container for executing the *ApplicationMaster* and the other 2 for executing tasks), it took about 61% and 75% of the makespan for Standalone with 128 MB and 64 MB RDD partitions, respectively. Similarly, Mesos took approximately 46% and 38% of the makespan of Standalone. Applications spend more time waiting for resources on the Standalone cluster manager than on YARN and Mesos. When the RDD partition sizes decrease, the number of tasks increases, reducing both the average waiting time and execution time of applications on Standalone. Moving from 64 MB to 32 MB partition size almost halved the makespan, surpassing YARN.

Under Mesos, reducing RDD partition size also has a large impact on the makespan, (35% and 23% for each reduction, respectively). For all the static resource allocation experiments, the Mesos cluster manager produces the best results, since Mesos implemented some internal fair sharing of resources.

Figure 11 compares the makespan of different configurations when using dynamic resource allocation without speculative execution. There is a substantial reduction in makespan values for all experimental setups. With a 128 MB partition size, the makespan was reduced by approximately 63% on Standalone, 34% on YARN and 28% on Mesos when compared to static resource allocation.

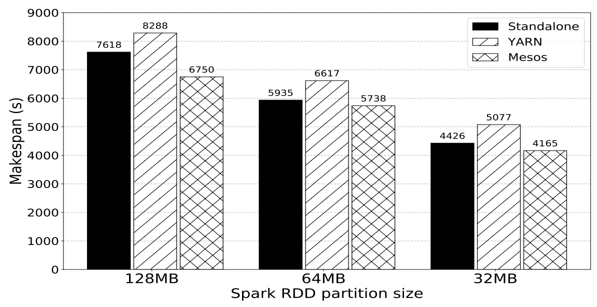


Figure 11: Makespan: dynamic/1 core/2 GB/no spec

With dynamic resource allocation, the first application submitted to all the cluster managers was allocated only the exact number of executors it needed, therefore, reducing the waiting time of subsequent applications, and hence a reduction in the total makespan. Similar to static resource allocation, the Mesos cluster manager produces the best results. The difference in makespan between Standalone and Mesos is very small on the configurations with 64 MB and 32 MB partitions. The relative performance of the cluster managers is unchanged, indicating that there is some overhead with YARN as with 32 MB partitions, all cores are busy doing "something" for a larger percentage of the makespan.

Impact of Speculative Execution. Figure 12 shows the makespan for static resource allocation with and without speculative execution of tasks enabled. Surprisingly, a small reduction in makespan values were observed, especially on the configuration with 32 MB RDD partition size, which has the largest number of concurrently running tasks, therefore, higher chances of speculation. The results show 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. In particular, on the configuration with 32 MB partition, the reduction in makespan is approximately 9% on YARN and less than 1% on Mesos. These values are lower than expected.

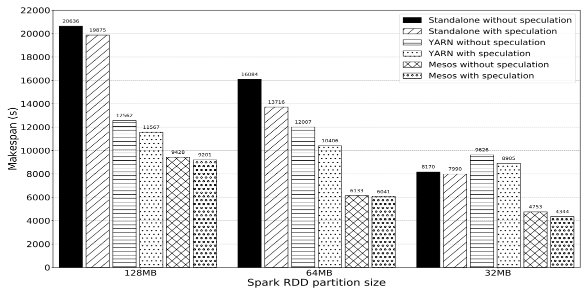
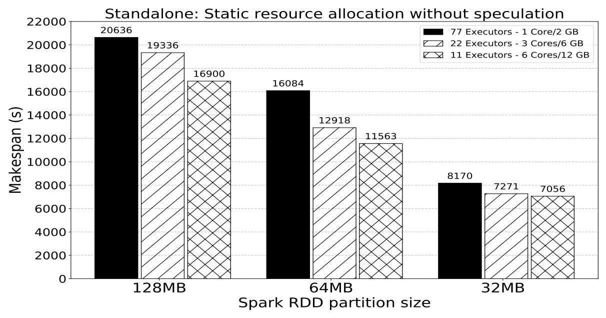


Figure 12: Makespan Speculation: static/1 core-2 GB

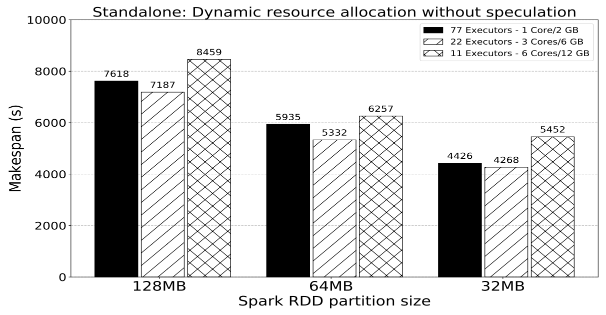
The low reduction in makespan while using speculation is related to the characteristics of the benchmark image processing applications and partition size used for the experiments. Many stages in the applications have long running tasks. Since re-launching tasks means starting them from the beginning 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.

Impact of Spark Executor Size on Makespan. Figure 13a shows that the makespan decreases with the increase in the number of CPU cores and memory per executor for Standalone in static resource allocation mode. This is because 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 benefits from this configuration, since small executors may increase the overhead of creating many JVMs and may lead to performance deterioration for such applications.

When dynamic resource allocation is enabled, the configuration with *6 cores/12 GB RAM* provides the worst performance on all the three partition sizes, as shown in Figure 13b. There are relatively few executors to allocate to applications compared to the other two configurations. If the number of tasks in a stage is not a multiple of the executor size, then some cores will be idle as an entire executor



(a) Standalone/static/no speculation



(b) Standalone/dynamic/no speculation

Figure 13: Makespan: various executor sizes

must be allocated atomically. With small numbers of tasks (in one case, 14), 4 cores must be idle (22% of the cores of 3 executors).

In general, there are many factors to consider in deciding optimum executor size. 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 node resources) to avoid resource contention, excessive garbage collection and executor failure which may negatively affect application 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.

Comparison with sequential applications. Finally, we compare the makespan of executing the distributed applications on Spark and equivalent sequential versions on a server equipped with large amount of resources. Dynamic resource allocation with 1 core/2 GB RAM per executor, 32 MB RDD partition size, and without speculative execution of tasks was selected for comparison in this section as it performed consistently strong. The makespan values are normalized based on the number of worker cores.

Figure 14 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 to the results from the sequential server. The results show we can achieve better performance by distributing the processing across multiple machines

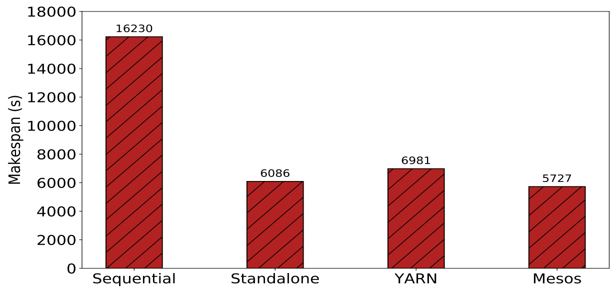


Figure 14: Makespan: *Sequential* and Spark with various parameter tuning (Cluster 1)

using Spark and tuning the various Spark configuration parameters, due to automatic parallelization and distribution of large-scale computations, concurrent execution of multiple tasks, optimized partitioning of the input data across the cluster nodes and independent disk I/O on the cluster nodes.

6 CONCLUSIONS AND FUTURE WORK

In this paper, we present a case study of Spark resource configuration and management on two different cluster environments using real image processing applications for plant phenotyping: *imageRegistration*, *flowerCounter* and *imageClustering*. In general, we show that selection of configuration parameters and resource managers is challenging, but permits the reduction of application execution time. In particular, we have shown that parallelism alone can provide great speedup, but there must be enough data partitions to keep all worker cores busy. Either more simultaneous applications can be submitted or the partition size decreased. The data we have currently collected does not meet the standard for true Big Data, but the large compute requirements of our initial benchmarks on the small datasets do enable Spark to improve performance. Optimizing performance by tuning the parameters still appears to require repeated measurements with various configurations.

For future work, we plan to investigate further the influence of more configuration parameters in a GPU-enabled Spark cluster. The initial versions of the applications did not use GPUs as the single server did not have a GPU. The cluster machines could not use their GPU as GPU virtualization is not fully supported in KVM at this time. The applications will have to be rewritten to use GPU functions and the cluster migrated to the physical machines or we must use a different virtualization infrastructure.

Furthermore, the use of container technologies with automation deployment frameworks such as Kubernetes is of future interest. We also intend to consider more interesting image processing applications that are more representative in that they cover a wide range of Spark's high level transformations and actions. The project will also incorporate various bioinformatics and genomics applications that can share the Spark/Hadoop infrastructure and may have differing resource requirements over time than those studied here.

ACKNOWLEDGEMENTS

We thank the Natural Sciences and Engineering Research Council (NSERC) of Canada for partial support for this project through Discovery Grants. This research was undertaken thanks in part to funding from the Canada First Research Excellence Fund.

REFERENCES

[1] Quan Chen, Minyi Guo, Qianni Deng, Long Zheng, Song Guo, and Yao Shen. 2013. HAT: history-based auto-tuning MapReduce in heterogeneous environments. *The Journal of Supercomputing* 64, 3 (June 2013), 1038–1054.
[2] Tatsuhiro Chiba and Tamiya Onodera. 2016. Workload characterization and optimization of TPC-H queries on Apache Spark. In *ISPASS*. Uppsala, Sweden, 112–121.
[3] Jeffrey Dean and Sanjay Ghemawat. 2008. MapReduce: Simplified Data Processing on Large Clusters. *Commun. ACM* 51, 1 (2008), 107–113.
[4] Christina Delimitrou and Christos Kozyrakis. 2014. Quasar: Resource-efficient and QoS-aware Cluster Management. In *ASPLOS*. Salt Lake City, UT, 127–144.
[5] Massimo Franceschet. 2011. PageRank: Standing on the Shoulders of Giants. *Commun. ACM* 54, 6 (June 2011), 92–101.
[6] Ali Ghodsi, Matei Zaharia, Benjamin Hindman, Andy Konwinski, Scott Shenker, and Ion Stoica. 2011. Dominant Resource Fairness: Fair Allocation of Multiple Resource Types. In *NSDI*. Boston, MA, 323–336.
[7] Benjamin Hindman, Andy Konwinski, Matei Zaharia, Ali Ghodsi, Anthony D. Joseph, Randy Katz, Scott Shenker, and Ion Stoica. 2011. Mesos: A Platform for Fine-grained Resource Sharing in the Data Center. In *NSDI*. Boston, MA, 295–308.
[8] Min Li, Jian Tan, Yandong Wang, Li Zhang, and Valentina Salapura. 2017. SparkBench: a spark benchmarking suite characterizing large-scale in-memory data analytics. *Cluster Computing* 20, 3 (Sept. 2017), 2575–2589.
[9] David G. Lowe. 2004. Distinctive Image Features From Scale-invariant Keypoints. *International Journal of Computer Vision* 60, 2 (Nov. 2004), 91–110.
[10] Matt Massie, Bernard Li, Brad Nicholes, Vladimir Vuksan, Robert Alexander, Jeff Buchbinder, Frederiko Costa, Alex Dean, Dave Josephsen, Peter Phaal, and Daniel Pocock. 2012. *Monitoring with Ganglia: Tracking Dynamic Host and Application Metrics at Scale.* O'Reilly Media, Inc.
[11] N. Nguyen, M. M. H. Khan, Y. Albayram, and K. Wang. 2017. Understanding the Influence of Configuration Settings: An Execution Model-Driven Framework for Apache Spark Platform. In *CLOUD*. Honolulu, CA, 802–807.
[12] Kay Ousterhout. 2014. Scripts to analyze Spark's performance. https://github.com/kayousterhout/trace-analysis.
[13] Kay Ousterhout, Ryan Rasti, Sylvia Ratnasamy, Scott Shenker, and Byung-Gun Chun. 2015. Making sense of performance in data analytics frameworks. In *NSDI 15*. Oakland, CA, 293–307.
[14] Ivanilton Polato, Reginaldo Re, Alfredo Goldman, and Fabio Kon. 2014. A Comprehensive View of Hadoop research-A Systematic Literature Review. *J. Netw. Comput. Appl.* 46, C (Nov. 2014), 1–25.
[15] Konstantin Shvachko, Hairong Kuang, Sanjay Radia, and Robert Chansler. 2010. The Hadoop Distributed File System. In *MSST 2010*. Washington, DC, 1–10.
[16] Animesh Trivedi, Patrick Stuedi, Jonas Pfefferle, Radu Stoica, Bernard Metzler, Ioannis Koltsidas, and Nikolas Ioannou. 2016. On the [Ir] relevance of network performance for data processing. In *HotCloud*. Denver, CO, 126–131.
[17] Vinod Kumar Vavilapalli, Arun C. Murthy, Chris Douglas, Sharad Agarwal, Mahadev Konar, Robert Evans, Thomas Graves, Jason Lowe, Hitesh Shah, Siddharth Seth, Bikas Saha, Carlo Curino, Owen O'Malley, Sanjay Radia, Benjamin Reed, and Eric Baldeschwieler. 2013. Apache Hadoop YARN: Yet Another Resource Negotiator. In *SOCC*. Santa Clara, CA, Article 5, 16 pages.
[18] Jorge Veiga, Roberto R. Expósito, Xoán C. Pardo, Guillermo L. Taboada, and Juan Touri no. 2016. Performance evaluation of big data frameworks for large-scale data analytics. In *(Big Data*. Washington, DC, 424–431.
[19] Aibishek Verma, Luis Pedrosa, Madhukar Korupolu, David Oppenheimer, Eric Tune, and John Wilkes. 2015. Large-scale Cluster Management at Google with Borg. In *Eurosys*. Bordeaux, France, 18:1–18:17.
[20] Hongbin Yang, Xianyang Liu, Shenbo Chen, Zhou Lei, Hongguang Du, and Caixin Zhu. 2016. Improving Spark Performance with MPTE in Heterogeneous Environments. In *ICALIP*. Dubai, UAE, 28–33.
[21] Matei Zaharia, Mosharaf Chowdhury, Michael J. Franklin, Scott Shenker, and I. Stoica. 2010. Spark: cluster computing with working sets. In *HotCloud*. Boston, MA, 1–7.
[22] Qi Zhang, Mohamed Faten Zhani, Yuke Yang, Raouf Boutaba, and Bernard Wong. 2015. PRISM: Fine-grained Resource-aware Scheduling for MapReduce. *IEEE Transactions on Cloud Computing* 3, 2 (Jan. 2015), 182–194.
[23] Xuechen Zhang, Ujjwal Khanal, Xinghui Zhao, and Stephen Ficklin. 2016. Understanding Software Platforms for In-Memory Scientific Data Analysis: A Case Study of the Spark System. In *ICPADS*. Wuhan, China, 1135–1144.