Improving Real Time Tuning on YARN

by

Víctor Pablo Navarro-Belmonte,

A thesis submitted to the
Faculty of Graduate and Postdoctoral Affairs
in partial fulfillment of the requirements for the degree of

Master of Computer Science with Data Science Specialization

Ottawa-Carleton Institute of Computer Science
Department of Computer Science
Carleton University
Ottawa, Ontario
November, 2017

The undersigned hereby recommends to the
Faculty of Graduate and Postdoctoral Affairs
acceptance of the thesis

Improving Real Time Tuning on YARN

submitted by **Víctor Pablo Navarro-Belmonte**,

in partial fulfillment of the requirements for the degree of

Master of Computer Science with Data Science Specialization

Professor Amiya Nayak, Ph D, External Examiner

Professor Michel Barbeau, Ph.D, Examiner

Professor Frank Dehne, Ph.D, Thesis Supervisor

Professor Mark Lanthier, Ph.D, Chair,
Department of Computer Science

Ottawa-Carleton Institute of Computer Science
Department of Computer Science
Carleton University
July, 2017

Abstract

Big data is becoming a significant part of the operations of modern organizations. Performing analysis of large amounts of data requires computer clusters to run the calculations and analysis. YARN is an internal framework that is responsible for coordinating big data jobs for some popular distributed storage and processing frameworks like Hadoop and Spark. Running YARN with the correct configuration parameters is critical for the good performance of a cluster. KERMIT is an online tuner of YARN configuration parameters that aims to improve cluster performance. The first KERMIT implementation proved the feasibility of the concept. In this study we modified the tuning algorithms inside the KERMIT components; by doing so, we achieved a reduction in the execution time as compared to industry benchmarks for a shallow tuning technique. We also verified that KERMIT can tune Spark; this suggests that Kermit could be used in other YARN-based frameworks.

This thesis is dedicated to my loving wife Karen, my parents Luz del Carmen and Victor Manuel, and my sister Luz del Carmen. Your unconditional support has helped me in every way during this incredible journey.

Acknowledgments

First, I would like to express my sincere gratitude to my advisor Dr. Frank Dehne, a Chancellor's Professor in the School of Computer Science at Carleton University. Thanks for all your advice and your guidance through these years of learning from you. I feel honored and fortunate to have worked under your supervision.

I would also like to thank Dr. Amiya Nayak, Dr. Michel Barbeau, and Dr. Mark Lanthier for reviewing this thesis. Furthermore, I want to thank Mikhail Genkin for sharing his extensive knowledge and experience regarding Big Data. Undertaking this project would not have been possible without his advice.

Also a very sincere thank you to the Research Computing Services of Carleton University, especially to Dr. Sylvain Pitre and Blake Henderson. The help your Department provided was essential for this project.

I also want to thank my fellow members on the research team, Maria Pospelova, Yabing Chen and Ali Davoudian. All your contributions paved the road for me to do my research and I feel lucky to have worked with you. Finally, I want to thank the Consejo Nacional de Ciencia y Tecnología (the National Council of Science and Technology of Mexico) (abbreviated CONACYT) for granting me a scholarship that covered my graduate studies.

Contents

List of Tables

List of Figures

Chapter 1

Introduction

1.1 Motivation and Background

Data science is a field closely related to Computer Science that shows a lot promise if it can be put to use by academic and industry leaders. The quantity of data that all organizations in our society generate is growing each year. Those large amounts of data have a lot of untapped potential, which, if it could be put to use, could result in significant financial gains. To process this new volume of data, the data scientist cannot rely on using a single computer but rather may require a cluster of many computers doing data processing. Yet Another Resource Negotiator (YARN) is at the heart of many applications that are used to do Data Science. This research work aims to provide the means that can result in performance gains for clusters using the YARN framework. The aim of the work presented in this thesis is to build a tool that will improve the performance of YARN applications by taking care of choosing the configuration values that yield the best performance.

1.2 Problem Statement Definition

YARN is a framework used in Big Data platforms where performance is a critical attribute. To achieve a good level of performance the hardware capabilities of the cluster that is running YARN is a very important factor. Also, the parameters used in the YARN configuration are very relevant. The difference in performance between a well-tuned YARN cluster and a poorly-tuned one is not trivial. This means that a set of good configuration values is something vital for every YARN cluster [3].

Unfortunately, getting the optimum configuration for a YARN cluster is not an

easy task [4]. The professionals that have the knowledge to tune a YARN cluster are rare and the cost of their services can be very high. This limits the number of organizations that can aim to set up a YARN cluster to do their data processing [2].

This proposed research aims to build on the previous work on a tool that functions as an automatic online tuner for YARN applications. Organizations will save both time and money if the barrier of having to engage an expert to do the manual tuning of a YARN cluster to obtain good performance is removed at least partially by the introduction of an automatic tool. Also, YARN deployment will be available to more organizations because of the reduced complexity of operation.

1.3 Scope of the Research Work

This work aims to build on previous work performed by my colleagues in the KERMIT team working under the direction of Dr. Frank Dehne. The objective of this project is to improve on the original version of KERMIT by adding better tuning capabilities, and it was not intended to create a new tuning tool from scratch. This research work was limited to efforts to tune YARN configuration parameters and did not attempt to do any tuning outside of YARN. The performance of the tuning efforts will be evaluated using known industry benchmarks; custom workloads to test KERMIT performance will not be created. This research work will focus on tuning two very significant parameters: the size of memory and the number of virtual cores assigned to the YARN containers; tuning other parameters could be considered for future work on KERMIT.

1.4 Contributions of this Research Work

The work presented in this thesis is based on the previous efforts of the team in charge of developing KERMIT [2], a tool that aims to do dynamic and automatic tuning of YARN parameters. The previous work showed that KERMIT had a firm foundation to become a solid tuning tool. In this work, KERMIT was improved by adding a new tuning strategy and better logging capabilities so we would be able to tell how the process was being executed. New experiments were performed where KERMIT tuning proved to give generally better performance than a shallow tuning option; these experiments were performed using larger data sets than those used

in the previous study. Also as part of the experiments it was demonstrated that KERMIT can also work with other frameworks that rely on YARN such as Spark. These results were presented in the 18th IEEE International Conference on High Performance Computing and Communications (HPCC 2016) [5].

1.5 Thesis Organization

First in Chapter 2 - Background information, basic information regarding the algorithms and frameworks used in this research work is presented. In Chapter 3 - Literature Review, there is a review of research work done by different authors that is relevant to this research. In Chapter 4 - Problem Statement, the objectives for this research work are defined. Chapter 5 - KERMIT Online Tuning Overview, elaborates on the design and inner workings of KERMIT, the tool that was used as the foundation of this research (part of this chapter was presented in HPCC 2016 [5]). In Chapter 6 - Kermit Enhancements, the changes that were made to KERMIT in order to expand and improve its functionality are presented (part of this chapter was presented in HPCC 2016 [5]). Those enhancements were tested, and in Chapter 7 - Data Analysis, the experiments that were performed and the results obtained are described (presented in HPCC 2016 [5]). In Chapter 8 - Conclusions, last thoughts regarding the new version of KERMIT are given and in Chapter 9 - Future Work, alternatives that we think are relevant to continue our work are reviewed.

Chapter 2

Background Information

2.1 Introduction

This chapter is partially based on three reference books: Hadoop: data processing and modelling [6], Learning YARN: moving beyond MapReduce [1] and Learning Spark [7].

Today, our society is generating data at a pace that is hard to evaluate. Almost every aspect of our existence implies big data generation and processing. Social network interactions, the purchase of goods, even our own bio-medical information is captured and stored. There is a new-found value present in these massive data collections. Organizations can now use all this data to gain insights that could lead to significant discoveries. Having all this information available for specialists to use offers a great opportunity that comes with some new challenges such as how to store and process such massive amounts of data. With the growth of data, there was also a major jump in the capabilities of tools that could process large quantities of data. The tools did not only become able to handle larger data sizes but also become capable of doing so at a much lower operational cost regarding hardware and software licenses. That is especially important because, before this new trend of Big Data, there were just a handful of organizations that had a need to process very large data sizes. That is not true any longer; small startups, and even individuals, can have operations that involve large quantities of data.

The tendency of having more and more data to process and analyze makes it almost impossible to use a single computer to perform these tasks. That is one of the reasons that led to the data mining systems failing to achieve widespread adoption; these systems were traditionally bound to the capabilities of a single computer. The

problem of coping with a significant increase in the data to process has been dealt with in two different ways:

Scale up: This means increasing the hardware capabilities of the computer running the application; this could mean an upgrade in some hardware components like RAM and CPU, or moving the application to another computer that uses the same architecture as the previous one. This is relatively easy to do, but can be very costly in financial terms; there is also a hardware limit for the gains that can be obtained by just scaling up.

Scale out: Instead of increasing the capabilities of the hardware running the application, this approach spreads the processing tasks to new computers thereby forming a cluster. On the one hand, this approach requires big changes in terms of software architecture because computer applications have traditionally been designed to run on a single computer; on the other hand, scaling out can result in lower monetary costs and more potential to grow beyond the current state of the art in CPUs capabilities [6].

2.2 Technologies Overview

Spark and Hadoop are projects supported by the Apache Software Foundation. These applications are intended to process massive amounts of data and they both use YARN (Yet Another Resource Negotiator) internally to manage and optimize their resources. A brief overview of the Hadoop, Spark and YARN technologies follows, as they will be the focus in this research.

2.2.1 Hadoop Overview

Hadoop is a programming framework that has its origins in two academic papers describing the Google File System (GFS) [8] and MapReduce [9]. These two technologies provided a platform that can process data on a very large scale. Those two academic papers were implemented by Doug Cutting to create the first version of Hadoop.

Hadoop was designed around the following architectural principles:

- Having the option to scale out by adding more nodes to the cluster.

- Having the ability to run on commodity hardware (not requiring expensive high-end hardware).
- Having the ability to identify failures and working around them.
- Providing transparent services to allow the user to focus on the problem the user is trying to solve.

Those architectural principles are present in the two main components of Hadoop HDFS and MapReduce.

2.2.1.1 HDFS (Hadoop Distributed File System)

HDFS is a non "Portable Operating System Interface (POSIX)" compliant file system that is designed to store very large data sets. It is designed to be able to spread the data among the disks of the nodes of a cluster; this is a direct implementation of the scale out principle.

- HDFS files are composed of individual blocks that typically are 64 MB in size; this is significantly different to most file systems that use a block size around 4 to 32 KB.
- The design for HDFS prioritizes throughput over latency; this means that a large file will be read fast, but a request to find many small files will be relatively slow.
- HDFS is optimized for workloads that will write data once and read the data many times.
- There is a process named DataNode that manages the HDFS files on every cluster.

2.2.1.2 MapReduce

MapReduce is a data processing paradigm that defines how the data will be input and output in two phases (Map and Reduce); this definition is used to process large data sets. MapReduce will try to optimize the processing by making the processing tasks run in the HDFS nodes that hold the data; the fundamental principle of MapReduce is that it is preferable to move computed data than to move data for computation.

MapReduce uses the divide and conquer approach to break down tasks and run them in a parallel manner [1]. The MapReduce user is responsible for defining the criteria for the execution of the Map and Reduce tasks; however, every other aspect of the actual execution such as coordination and parallelization are a responsibility of the MapReduce implementation. The end result is that the actual size of the data and the cluster should be transparent to the user and the only changes should be in the execution time.

The high-level architecture of the MapReduce framework consists of three modules:

- MapReduce API: The end user API (application programming interface) used by the developers to program MapReduce Jobs to be executed.
- MapReduce Framework: This is the runtime implementation of many phases in a MapReduce job such as map, sort/shuffle/merge aggregation, and the reduce phase.
- MapReduce System: The back-end infrastructure required to run the MapReduce application, manage resources and schedule jobs.

The MapReduce system is composed of two components: JobTracker and TaskTracker.

JobTracker

- JobTracker: the master daemon within Hadoop that is responsible for resource management, job scheduling, and management.
- Hadoop clients link with the JobTracker to submit or kill jobs and poll for a job progress.
- JobTracker validates the client request, and if validated it allocates the TaskTracker nodes for the execution of MapReduce tasks.
- JobTracker monitors TaskTracker nodes and their resource utilization, that is, how many tasks are currently running, and the count of MapReduce task slots available; it also decides whether the TaskTracker node needs to be marked as a blacklisted node, and so on.

- JobTracker monitors the progress of jobs, and if a job/task fails, it automatically reinitializes the job/task on a different TaskTracker node.

TaskTracker

- TaskTracker: a daemon that runs on every node responsible for the execution of MapReduce tasks. A TaskTracker node is configured to accept a number of MapReduce tasks from the JobTracker daemon [1].
- TaskTracker creates a new Java Virtual Machine (JVM) process to perform the MapReduce logic. Running a task on a separate JVM helps ensure that failure of the task does not harm the health of the TaskTracker daemon.
- TaskTracker monitors these JVM processes and updates the task progress to the JobTracker regularly.
- TaskTracker also sends a "heartbeat message" (a periodic message to confirm that the TaskTracker is still working) and its current resource utilization metric to the JobTracker every few minutes.

2.2.2 YARN Overview

YARN (Yet Another Resource Negotiator) is a resource management framework meant to improve on the shortcomings of MapReduce.

2.2.2.1 MapReduce Limitations

The most important limitations of MapReduce are:

- Unavailability and unreliability: The JobTracker is a single point of failure. If the process is killed, then the job is stopped, and all the information for queued jobs is lost.
- Batch processing only: There is no support for anything that does not fit the MapReduce paradigm.
- Nonscalability and inefficiency: There is a complete dependency on the master daemon. If the cluster grows too big, then the single JobTracker will struggle to manage efficiently all the nodes efficiently; this will result in underutilized resources and bad performance.

- Partitioning of Resources: The configuration for a cluster had to partition the resources allocated into map or reduce task slots, that kind of partitioning resulted in a partial utilization of the cluster resources.

To try to address these problems, YARN was designed in a way that splits the responsibilities of the JobTracker into three separate parts: management, job scheduling, and job monitoring. YARN is composed of three main components, with each taking care of one of those responsibilities, this is illustrated in Figure 2.1.

2.2.2.2 ResourceManager

A ResourceManager is a per cluster service that manages the scheduling of resources for applications. It maximizes cluster utilization in terms of memory, CPU cores, fairness, and SLAs (service-level agreements). To be able to cope with different policy constraints, it has algorithms in terms of customizable schedulers for factors such as capacity and fairness that allows for resource allocation in specific ways.

ResourceManager is composed of two main components:

- Scheduler: A pluggable component that is only responsible for resource allocation to applications; the criteria for deciding how the resources will be allocated depends on the specifics for every scheduler.

- Applications Manager: A service used to manage application across the cluster. It is in charge of accepting new applications submissions and for providing resources for applications to start; it is also responsible for monitoring the application and restarting the application in the event of a failure.

2.2.2.3 NodeManager

The NodeManager is a worker service that runs on every node and is responsible for the execution of containers based on the node capacity regarding memory and CPU cores. There is a direct communication of status messages with the ResourceManager in the form of a heartbeat signal.

2.2.2.4 ApplicationMaster

The ApplicationMaster is a per application, framework-specific library that manages each instance of an application that runs within YARN. It is treated by YARN as

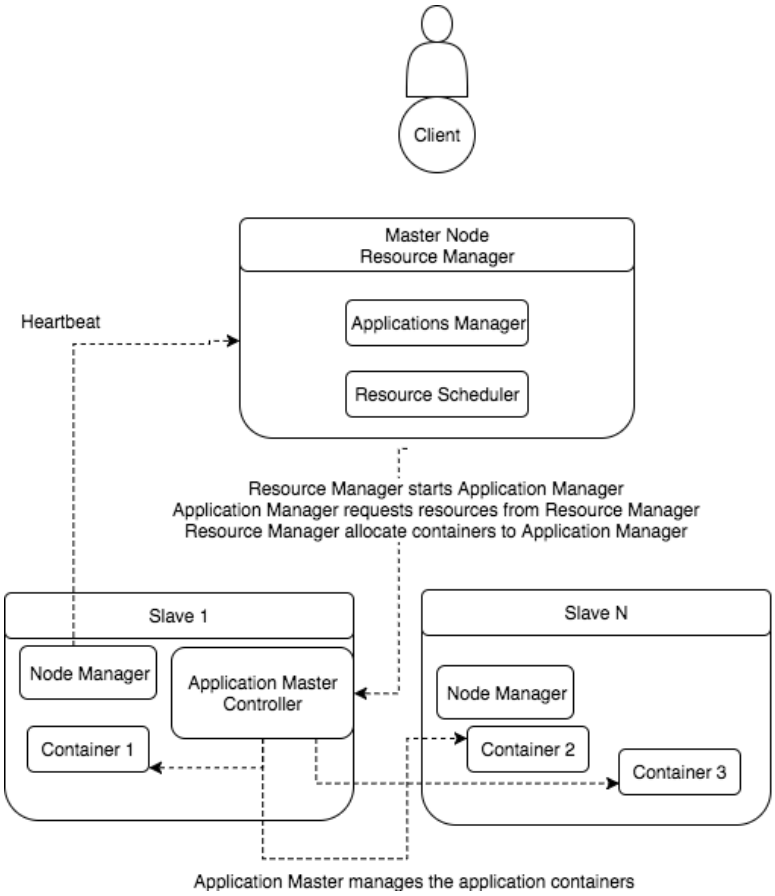

Figure 2.1: Overview of the YARN components [1].

a third party component that negotiates resources with the ResourceManager and interacts with the NodeManager to execute tasks. The ResourceManager allocates containers to the ApplicationMaster and those containers are used to run the application. The Application master is also in charge of monitoring the status of the containers and notifying the ResourceManager when containers are finished.

2.2.2.5 Containers

The container is a logical bundle of resources in terms of memory, CPU, disk space and so on that is bound to a particular node. The ResourceManager scheduler service dynamically allocates resources as containers. A container grants rights to an ApplicationMaster to use a specific amount of resources of a particular host. An ApplicationMaster is considered as the first container of an application and it manages

the execution of the application logic on allocated containers.

2.2.2.6 How YARN Improves on MapReduce

In a previous section the most significant problems with MapReduce were noted, in this section the way YARN addresses these issues will be discussed:

- Scalability and higher cluster utilization: In YARN the responsibility for resource management, job scheduling, and monitoring is divided among separate daemons, allowing YARN daemons to scale the cluster without degrading its performance of the cluster. With a flexible and generic resource model in YARN, the scheduler can handle the overall resource profile for specific types of applications. This structure makes the communication and storage of resource requests efficient for the scheduler, thus resulting in higher cluster utilization.

- High availability of components: Fault tolerance is a core design principle for YARN. Responsibility for ensuring a high availability of components is delegated to ResourceManager and ApplicationMaster. The application-specific framework, ApplicationMaster, handles the failures of containers. The ResourceManager handles the failures of NodeManager and ApplicationMaster.

- Flexible resource model: In MapReduce, resources are defined as the number of map and reduce task slots available for the execution of a job. In YARN, a resource request is defined in terms of memory, CPU, and so on. This results in a generic definition for a resource request by an application.

- Multiple data processing algorithms: The MapReduce framework is limited to batch processing only. YARN is designed to allow for the need to perform a wide variety of data processing using the data stored in HDFS drives. YARN is a framework for generic resource management and it allows users to execute multiple data processing algorithms on their data.

2.2.3 Spark Overview

Spark is an open-source cluster-computing framework that was started in 2009 as a research project in the UC Berkeley RAD Lab (Reliable, Adaptive and Distributed Systems Laboratory); this was later transformed into the AMPLab (Algorithms, Machines and People Lab). The members of that team had previously worked on Hadoop

MapReduce and noticed that MapReduce was not efficient for interactive and iterative workloads. The team tried from the beginning to make Spark efficient for interactive queries and iterative algorithms, incorporating ideas like in memory storage and efficient fault recovery. Some research papers were published about Spark, and it showed gains in speed of around 10-20 times in comparison to Hadoop for certain jobs [7].

Spark was first used by UC Berkeley researchers but soon after it was adopted by many organizations. Besides UC Berkeley some other organizations have contributed heavily to Spark development such as Databricks, Yahoo! and Intel.

In 2011, the AMPLab began to develop the higher level parts of Spark such as Shark and Spark Streaming; these components and others are sometimes referred as the Berkeley Data Analytics Stack (BDAS). Spark was open sourced in March 2010, and was transferred to the Apache Software Foundation in June 2013.

Spark is designed to be fast and general-purpose; it extends the MapReduce model to support more types of computation efficiently. One of the main features of Spark is its ability to run computations in memory to avoid the IO wait times associated with disks; however, it is also more efficient than MapReduce for complex applications that are running on disk. Spark is also designed to be flexible and accessible by offering straightforward APIs in Python, Java, Scala, and SQL.

2.3 Performance Tuning for YARN

Previous research has shown that the performance of YARN MapReduce applications depends on the values of their configuration parameters (whose performance related ones number more than 70) [10] [11]. Tuning these parameters requires an understanding of the characteristics of the application, the data and the system resources [4]. Also, there are complex inter-dependencies among the configuration parameters, which means that changing the value of one configuration parameter can have an enormous impact on the other configuration parameters [12]. Therefore, tuning configuration parameters is beyond the scope of knowledge of traditional enterprise IT staff, which makes automatic parameter tuning highly desirable.

There are two kinds of strategies for tuning of YARN parameters: manual and automatic.

2.3.1 Manual Tuning

The manual tuning strategy involves a person that has the technical expertise to tune YARN clusters; this person normally assesses the current cluster configuration and application needs to come up with initial configuration parameters. The configuration is used for some trial runs and metrics are collected; the metrics are analyzed by the technical expert who then tweaks the configuration and reruns tests to see the results. This cycle is sometimes repeated until the expert is satisfied with the results obtained.

Obviously, this kind of tuning is constrained by the availability of people that have the expertise to perform these tasks. Currently, there is substantial market demand for individuals with this kind of expertise. This demand has resulted in high hiring costs for such professionals - which can lead to a heavy financial burden for any company interested in using YARN for data processing. It is noted that many of these experts can base some of their decisions on available Rule of Thumb (RoT) tuning guides [13–15].

Another downside to this kind of tuning is that the parameter configuration remains static after the expert finishes the tuning job; this means that modifications in terms of cluster size, network configuration or job characteristics may change the performance of the cluster. This could make another round of manual tuning necessary.

2.3.2 Automatic Tuning

Automatic tuning consists of getting a program to process and analyze the information related to the performance of a cluster; this will result in configuration changes that aim to increase the performance of the cluster. There are currently two types of automatic tuning: static and dynamic.

2.3.2.1 Static Tuning

A static tuning strategy first comes up with a configuration based on a default setting or a rough understanding of application characteristics. Next, some test runs of the application are executed with enabled profiling, and data such as job performance counters, system monitoring logs, and profiling outputs are collected. The user then feeds the results to a performance analyzer or manually analyzes the statistics and generates a new configuration. The process is usually repeated for multiple runs

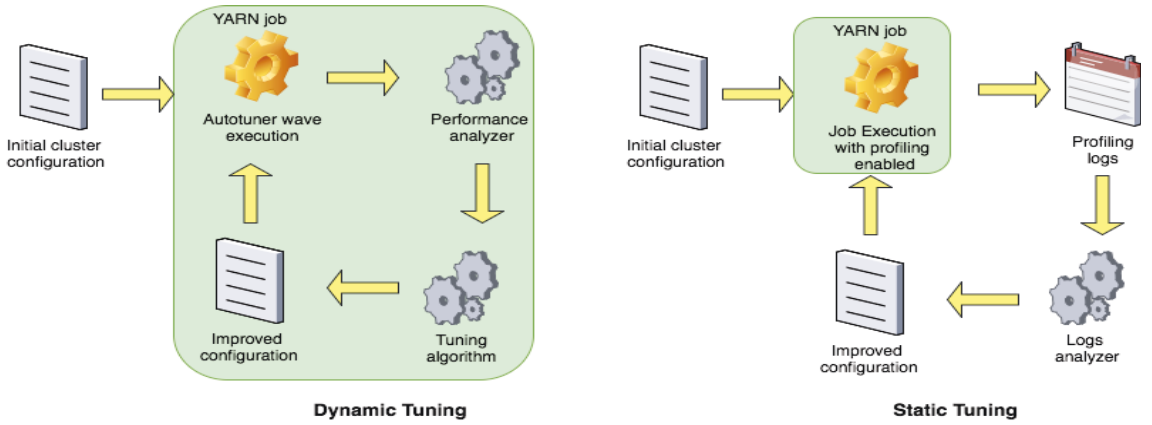

Figure 2.2: Dynamic vs. Static Hadoop/YARN parameter tuning

until the desired performance level is reached. The selected configuration is then employed while running the application on production clusters. This strategy has some drawbacks. First, it is time consuming especially when it requires many test runs and applications that involve long running jobs. Second, for applications which only run a few times or perhaps just once, static tuning of parameters is not cost-effective. Third, as mentioned previously, the optimal YARN configuration not only depends on the characteristics of the application but also on the size and the contents of the associated input data and on the cluster hardware configuration. Therefore, each change in those previously mentioned times should ideally result in an adjustment of the parameters. Fourth, with regards to probable data non-uniformity (skew), different tasks inside a job may need a different amount of resources as they process different sizes of data. In other words, no single configuration fits all the tasks inside a job. Finally, handling the dynamic characteristics of jobs and cluster features through static tuning strategies may cause performance degradation [11]. The static tuning process is represented in the Figure 2.2.

The aforementioned drawbacks of static tuning strategies caused the emergence of dynamic online performance tuning methods which aim at monitoring the execution of a MapReduce application and adjusting the associated performance-tuning parameters using the statistics gathered.

2.3.2.2 Dynamic Tuning

A dynamic tuning approach (also called online tuning) aims to tune the YARN clusters by responding to the real time changes in performance. This approach relies on having a pluggable component that can monitor in real time the performance metrics inside the cluster; this information will be analyzed by a tuning engine that will decide whether or not it is necessary to make changes to the parameters currently used by the YARN cluster. If changes are deemed appropriate for the given conditions, then a message with the new parameters will be sent to another pluggable component that has been specifically designed to change the configuration parameters on the fly without the need of restarting the cluster or changing configuration files. The differences between a dynamic and a static tuning strategy can be summarized in the Figure 2.2.

All the necessary metrics and statistical information will be collected and analyzed in order to evaluate if the parameter change resulted in performance improvement. The internals of the tuning algorithm will decide if additional tuning is appropriate or if the current configuration parameters should remain unchanged.

It must be noted that the specifics on how to perform this will vary greatly with different teams and projects; different algorithms and parameters will be used, and results may differ because of data size or hardware restrictions.

Chapter 3

Literature Review

3.1 Introduction

In this research work, the focus will be on improving the performance of YARN clusters by creating a piece of software that will set appropriate parameters for the YARN configuration. Doing this also involves an optimization problem because YARN performance is going to be treated as a black box problem. In this , the main focus will be on reviewing the existing literature on tuning YARN and Hadoop parameters to get performance improvements; also reviewed will be some optimization methods for black box problems that could be applicable to the research. While the main topic of this thesis is YARN automatic tuning, it was also considered it would be useful in this research to evaluate the feasibility of using different optimization methods.

3.2 Improving Performance of Hadoop/YARN

In this section the efforts made to tune Hadoop/Yarn in a static manner will be described; by using the word static we mean that the configuration parameters remain the same during the execution of a job and the adjustment in parameters to improve performance requires a restart of the Hadoop processes.

One of the early attempts at doing automatic tuning in Hadoop jobs was by Schaefer and colleagues [16] who designed and built an automatic tuning language for MapReduce called Atune-IL. That language offers the user the chance to explore and define operative ranges for the values of some pre-selected tunable variables such as the number of map and reduce jobs. Atune-IL also allows a user to explore the potential performance impact of alternative code block implementations. This solution does

automatic tuning to some extent. It helps reduce the search space and contributes to handling the optimum parameters search by using a streamlined tool. However, Atune-IL has some areas that may discourage some potential users; for instance, the user needs to have some Hadoop expertise to be able to select important variables, and it also requires the user to learn a new syntax unique to this solution. It is also important to note that this solution was conceived before Hadoop 2.0 was released alongside YARN; with no YARN support, this solution can only work in a purely MapReduce setting.

Christopher Re, introduced Manimal [17], a fully automatic optimization framework that uses static code analysis to detect MapReduce program semantics. The authors combined widely-practiced Relational Data Base (RDB) tuning techniques with the MapReduce model, resulting in a 63% performance gain on selection operations. Their research suggests that similar tuning success could be achieved with other operations, such as projections and data compression. This research was not based purely on Hadoop or YARN; the researchers built a framework that was similar to Hadoop MapReduce but worked in a different manner because of the optimizations they added.

Another way of performing automatic tuning of Hadoop/YARN is to incorporate tuning algorithms into the Java compiler. An automatic tuning compiler for MapReduce, Panacea, was proposed in 2012 by Liu and colleagues [18]. Panacea is able to optimize MR parameters with little or no input from the developer. It automatically tuned a fixed subset of tunable parameters. Parameter value limits were either set by the user or auto-determined by Panacea. Parameters were varied independently. The Panacea authors concluded that this was a sub-optimal solution due to the complexity of the search space landscape. Liu et al. claimed up to a three-fold performance improvement was achieved without user intervention.

In 2013 Chen and colleagues introduced the solution they called HAT, a history-based automatic tuning framework for MapReduce [19]. HAT tunes the weight of each stage of a map and a reduce task according to the values of the tasks in the history. HAT orchestrates the Hadoop backup-task execution according to current and historical weights of the tasks. The authors claimed a 37% job execution time improvement. This approach obviously requires the prior generation of historic data in the cluster to be in a position to obtain a tuned set of parameters.

There is also a cost-based Hadoop automatic tuning framework called Starfish;

it was developed at Duke University by Babu and colleagues [3, 20]. The authors succeeded in applying a cost-based approach popular in relational databases to optimizing MapReduce performance. Starfish takes into account the different stages of a MapReduce program. It adjusts the tuning at various decision points, which include provisioning, optimization, scheduling, and data layout. The heart of Starfish is the What-If Engine. It employs a combination of simulation and model-based estimation to come up with ideal settings for Hadoop MapReduce tunables.

Another project that worked on the tuning of Hadoop jobs execution is called Gunther [21]; it was proposed by Liao and colleagues in 2013. This project treats the Hadoop parameters as a black-box optimization problem; the team evaluated some global search algorithms, and decided to implement their own version of a genetic algorithm (GA) and to use it as the search algorithm for the configuration parameters. Two clusters were used to evaluate this approach; the team obtained experimental results that showed that Gunther achieved near-optimal performance in a small number of trials (<30), and that it yielded better performance improvements than rule-of-thumb tuning and cost-based automatic tuning approaches. It must be noted that this solution was based purely on Hadoop MapReduce and it did not deal with the new YARN architecture. Besides this, the solution is also of a static nature; this means that many runs are required to generate the information needed by the GA to calculate the optimal configuration.

Another example of the heuristic approach published in the same year as Gunther was research work conducted by Bezhad and colleagues. Their efforts were entirely focused on HDFS performance; this work was not based on Hadoop or YARN but it is relevant because they demonstrated that a GA-based parameter search combined with dynamically intercepted HDFS calls could result in an up to 100-fold I/O write speed up for some tasks [22].

There was also an attempt at developing a machine-learning approach that was explored by Yigitbashi and colleagues [23]; the team assessed various machine-learning-based performance models. Their analysis was conducted on two common Hadoop applications, Terasort and WordCount, using datasets of various sizes. The authors concluded that support-vector regression exhibited the best performance among the machine learning methods. According to the authors, the support-vector regression-based automatic tuner was shown to outperform Starfish. The experimental design

for this research work consisted of running jobs with different parameter configurations to collect performance data. The performance data was fed to the predictive model to attempt to predict the best configuration.

In 2014 Filho and colleagues worked on a search-based evolutionary computation approach to tune MapReduce [24]. They proposed an adaptive tuning mechanism that enabled the assigning of specific resources to each job within a query plan. A data structure mapping a job to tuning solutions was created based on an analysis of the source code and log files; rules of thumb were used to build the parameter configurations based on the characteristics of the jobs.

3.3 Dynamic Hadoop/YARN Auto-tuners

MRONLINE [25] is a dynamic parameter auto-tuner which allows multiple task-level configurations (during an application execution) instead of a single application-level configuration which is by default supported by YARN. It includes a tuner component implementing the SHC (Smart Hill Climbing adaptive search) method in order to explore the configuration space, generating optimal or near optimal sample configurations, and assigning each configuration to a randomly selected task from task queue. MRONLINE dynamically monitors per-task statistics (such as CPU, memory and I/O utilization), and feeds them to the the dynamic tuner in order to estimate the cost of each configuration. It also uses a number of tuning rules which aim at increasing the speed of convergence which is especially beneficial when it is faced with jobs with a small number of tasks as they restrict the SHC explorer from achieve a global or near global optimum. MRONLINE testing deals with variance coming from I/O bottlenecks, network congestion or file system issues by repeating (four times) the experiments and using the average value from that set of tests [11]. MRONLINE performs its tuning task in a tuning wave pattern such that in $wave_i$, some sample configurations are assigned to the tasks belonging to this wave; their performance is estimated and fed to the online tuner. Then more sample configurations are generated and assigned to tasks in $wave_{i+1}$. However, this wave pattern slows down job execution as the launching of new tasks belonging to $wave_{i+1}$ is held off until finishing all the tasks belonging to $wave_i$. Their result shows at most a 30% performance improvement on a 19-node cluster with a suite of five representative MapReduce jobs.

Another dynamic auto-tuner [26] aims at tuning a Hadoop parameter called Concurrent Container Slot (CCS), which has a significant impact on its performance. This parameter determines the quantity of concurrent containers that can run on a node. With this tuner, there is no search algorithm; instead, there is a feedback controller that monitors for performance statistics linked to CCS that can give hints of performance issues occurring such as I/O and CPU bottlenecks. In this research project three types of controllers that followed different strategies to control parallelism in a node that were tested in this research project: (1) Water-level controller, (2) Proportional-derivative controller, and (3) Proportional-derivative pruning controller. The controller that exhibited the best performance in the tests conducted is the Proportional Derivative Controller (PD controller).

There was a mechanism in place to change the CCS parameter value in response to bad performance metrics and to increase or decrease the number of containers running on a certain node. The change in the CCS parameter happens when there is a certain number of containers. YARN was tweaked to support pausing and resuming containers to be able to increase or decrease the parallelism inside a node without having to wait for containers to finish their current task. This modified version of YARN can lower parallelism by suspending containers or it can increment parallelism by resuming suspended containers. If there are no suspended containers, it can create new ones. The results of the tests concluded that all the controllers improved the performance comparing against the best practice static value for CCS by up to 28%, the team concluded that this indicated the need for more parameters to be automatically tuned to get the best performance from a cluster.

3.4 Parameter Tuning and Adaptive Search Methods

In this section, approaches that could be used to search for the optimum configuration for certain parameters will be reviewed. First, a summary of the many aspects these methods share in common is presented; thereafter, the most relevant methods are discussed individually.

3.4.1 General Overview of Adaptive Search Methods

Assuming a given number of tunable parameters for a system, its performance P (also known as objective, cost or fitness) is considered as a function f of the parameters. That is, $P = f(C)$, where $C = (X_1, \ldots, X_N)$, is a configuration instance and P is measured as a single-dimension metric such as throughput, response time, system utilization, or a combination of these. By defining R_i as the parameter range for parameter X_i, and $R = R_1 \times \cdots \times R_N$, as the set of possible configurations, parameter tuning problem aims at finding the optimal configuration C^* which achieves the best performance, i.e.:

$$f(C^*) = \mathcal{MIN}_{C \in R} f(C)$$

However, the objective function f is often unknown in practice and it must be estimated through experimentation or simulation (i.e., a black-box optimization problem). In addition, for a large configuration space R, estimating the objective function f for each individual configuration C, can be costly and time-consuming [27]. These challenges can be tackled using adaptive (or stochastic) search methods which aim at solving hard combinatorial problems. Notice that these methods are stochastic as with the same input their solutions may be different. These search strategies usually include two phases:

1. A global search phase which aims at exploring an exponentially large parameter space to find a promising area S where the probability of finding the globally optimal configuration is greater than in other areas.

2. A local search phase which exploits S and local information to find the optimal (or near to optimal) configuration. In order to escape of getting stuck in locally optimal areas, global and local search phases are alternated [28] [29] [30]:

In a more precise way, an adaptive search algorithm mostly uses a framework that includes the following steps:

- **Exploration**: This step explores the whole configuration space (using a sampling method φ) with the goal of identifying good areas or regions which are good candidates to contain the global optimum.

- **Exploitation**: Using the local information (e.g., gradients) obtained from the previous phase, this step is used progressively to generate new solutions which are better than the existing ones.
- **Restart**: This step rewinds the algorithm through re-sampling from the whole configuration space. More precisely, by repeating the exploration phase, this makes it possible to generate candidate solutions far from the current solutions, which creates an opportunity to escape from regions containing locally optimal configurations. However, it can be a waste of processing effort as many new solutions can be far from the global optimum. The optimal balance between the number of exploration and exploitation phases in order to quickly converge to the global optimum requires a higher level of optimization and remains an open problem.

3.4.2 Adaptive Search Methods

The choice of the sampling strategy (φ) is the major difference among adaptive search methods. In the following we investigate some of the adaptive search methods which have been used in Hadoop/YARN configuration parameter tuning:

- **Hill Climbing (HC)** or Valley Descending [31] is a relatively simple, fast and popular gradient-based heuristic local search method which starts with a random candidate configuration C_0 as the trial solution and attempts to improve it steadily and gradually. More precisely, in iterative steps, by performing a slight modification to the trial solution C_i, a set of successor configurations are generated belonging to its neighborhood. Then a configuration C_{i+1} is selected from the neighborhood and, if it is better than C_i with regard to maximizing (or minimizing) the performance $P = f(C)$, it replaces C_i. The process continues until the existing solution is better than all the candidates in its neighborhood or an iteration threshold is reached. HC does not necessarily guarantee to find the global optimum as it just considers the neighboring configurations (local search) and the final solution configuration largely depends on the initial trial solution. In other words, it is optimal for uni-modal functions; however, it may get stuck in local optima for multi-modal functions [32].

  There are variants of HC that take into account the strategy used for selecting the next configuration, namely: simple hill climbing which selects the first

neighbor better than existing solution; steepest gradient hill climbing which selects the best neighbor in order to speed the convergence toward the optimum; stochastic hill climbing which randomly selects a neighbor, and random restart hill climbing which is a stochastic iterated version of hill climbing where the method is applied several times with different initial candidate configurations. This additional iteration increases the chance of finding the global optimum through augmenting the probability of attempting all the function hills, which is beneficial for multi-modal functions [33]. The strategy efficiently solves the N queen problem even for a large number of queens [34].

- **Recursive Random Search (RRS)** [35] first generates a number of random sample points in the search space to find promising areas. A promising area is then sampled recursively in the neighborhood of a sample point with minimum cost. Each time that sample point is improved, it is replaced by the new improved point. If no improvement is made after a finite number of repetitions, the neighborhood is shrunk. This process continues until the neighborhood reaches a minimum size, after which the local search is abandoned. RRS then restarts global random sampling to find better areas based on the results of the last iteration, and repeats the recursive search [36]. However, due to naive random sampling, the restarts may waste effort and make the process inefficient. Also, there is no guarantee of achieving a global optimum or near-optimum [37].

- **Simulated Annealing (SA)** [38] is a probabilistic search method based on the Metropolis Monte Carlo model [39], to simulate energy levels in cooling solids. More precisely, SA is analogous to annealing in metallurgy where heating (creating significant randomness in the movement of atoms) and slowly cooling (gradually leading to less randomness) brings materials to a state with well-organized crystals. Accordingly, this search method starts with a random trial solution C_0, and then generates a new solution C_1 from its neighborhood. If the new solution is superior to the existing one ($\Delta f = f(C_1) - f(C_0) < 0$), then C_1 is accepted to enable hill climbing (or valley descending) and used as the new solution. Otherwise, it may be accepted (to avoid getting trapped in local optima) or rejected depending on an acceptance probability function P(acceptance) = $e^{-\Delta f/t}$ where t is the current temperature in annealing. This process then continues from the new solution.

By the initial setting of the temperature at a high level, as in annealing, almost all moves will be accepted, and the search space is explored as widely and in as diverse a manner as possible, ignoring minor deteriorations in the quality of the solutions obtained (i.e., hills or local optima). Then the temperature gradually decreases. Consequently, there are three main parameters involved in directing the search in order to improve the solution quality: the initial and final temperature values, as well as the rate of changing the temperature; which will affect the speed of optimization [40]. It is worth mentioning that choosing a very high initial value for temperature t means SA makes too many uphill moves for one to be accepted; on the contrary, a very low initial value makes the search to drop into a local optimum without a possibility to escape from it. Thus, an optimum initial temperature must be somewhere between these two extremes [41]. SA suffers from slow convergence and a high computational load, which would be overkill for cases with few local optima [42].

- **Genetic Algorithms (GA)** [43] are based on ideas from natural selection, genetics, and evolution. GA starts with an initial set of randomly created individuals or solutions called a population. Each individual within the population is represented as a DNA chromosome by translating it into a long array of bits. These chromosomes evolve through iterations named generations. This evolution is performed through a combination of genetic operators, mainly selection, crossover, and mutation. In addition, the applicability of each individual or solution is evaluated through a fitness measurement function. Each new chromosome (or offspring) is usually created by selecting its parents from the existing population and by using the methods of merging or recombining chromosomes and also by applying crossover operations and/or mutating the offspring where a mutation randomly changes part(s) of the chromosome (by inverting some bit(s)) to introduce a new one to the population [40].

  In fact, the crossover operation explores local optima in the search space, while the mutation operation helps to break out from spurious solutions which are far from the global optimum, so that the GA is able to search for solutions globally. The evolution cycle will be stopped when the desired fitness level is reached, or if a certain number of rounds have been reached. By increasing the probability of selecting fitter parents, new chromosomes gradually get better. One must also take into account that a too low mutation rate may cause this method to

get trapped in local optima. On the contrary, a too high mutation probability would transform GA into a random search method [44].

- **Tabu Search (TS)** [45] uses a short term memory, or taboo list, in order to ensure that the search does not return to the optimum after the algorithm forces it out in a new direction. More precisely, during the optimization process, the short term memory stores recent solutions which means they cannot be used as new candidate solutions [46]. However, if the taboo memory is too long, this could slow down the process of optimization, or prevent optimization altogether [40].

- **Particle Swarm Optimization** uses the concept of a swarm of particles traveling through the search space and measuring the value that the explored function returns for the values represented by each particle. The information gathered by each particle guides the whole swarm to step by step travel to find an optimum. The execution of the algorithm finishes when all the particles travel to the same area; this area can be considered an optimum. The sampling for this algorithm can be done in different ways, for example the Random Sampling in Variable Neighborhoods (RSVN) that prevents premature convergence that leads to getting stuck on local optima [47] and a variant of Optimal Computing Budget Allocation (OCBA) that makes the algorithm more noise resilient [48].

These methods can be used to search for an optimal configuration for YARN, but there is particular need to be careful with the number of samples a method needs to reach a final value. It could be possible that it is preferable to use a method that is not the most precise in terms of reaching the absolute optimal configuration but on the other hand, it requires just a few samples. This will result in less exploration overhead and a better finishing time than a method that yields the best possible configuration but one can just use it for a short time because most of the time was used in the search for that configuration.

Chapter 4

KERMIT Online Tuning Overview

4.1 Introduction

In this chapter, previous work that has been done on the KERMIT project will be discussed. This has been a continuous team effort led by Professor Frank Dehne.

The team working on KERMIT has been composed of past and present grad students; this is the list of team members in no particular order:

- Frank Dehne.
- Mikhail Genkin.
- Maria Pospelova.
- Yabing Chen.
- Victor Pablo Navarro-Belmonte.
- Siyu Zhou.

It is important to note that KERMIT is an ongoing team project; the author of this thesis acknowledges this and notes that the contribution of every member has been essential to the current achievements of the project. A significant part of the research the author worked on takes as a base the previous efforts and breakthroughs achieved by other members of the team.

The sections in this chapter will review what the KERMIT project is; how Kermit works internally; what milestones were achieved by other members of the team, and what the challenges are that still have not been solved. Some of this chapter information has been presented in HPCC 2016 [5].

4.2 Design and Architecture

This section will be a brief overview of how YARN works internally inside Hadoop and how the KERMIT autotuner works inside YARN to monitor the internal performance and look for a parameter configuration that maximizes the performance of the job.

4.2.1 Hadoop Code

Hadoop is written in Java and also some small parts in C, and it makes use of some native command-line tools. Java was used to achieve high interoperability with fewer constraints related to operating systems (because of this Hadoop can easily run in Windows or Linux) and the C-native parts are used because they offer high performance in critical tasks.

Hadoop was originally designed with two main components, MapReduce and HDFS (Hadoop Distributed File System). HDFS dealt with the persistent storage and management of information; MapReduce was in charge of doing the actual data computation. These components were designed with a master/slave architecture and provided a good deal of robustness that comes from information not being stored in a single location making Hadoop resilient to hardware or network failures.

This approach exhibited some weak points, for instance the fact that map slots and reduce slots were statically assigned in the nodes; this resulted in work only being able to be performed in the slots that matched the current task being executed. This could lead to situations where there were not enough map slots to hold all the computational needs, and there would be reduce slots sitting idle. Another disadvantage was that the master node hosts the JobTracker and this was the single point of failure; all the applications were coordinated by the same entity, and this put a limit on the reliability and performance of the whole cluster [See figure 4.1]. These issues were worked out in the next Hadoop versions that introduced radical improvements.

Hadoop has experienced major changes in its design since Hadoop 2.2.0; the most important change was the introduction of another MapReduce programming model: Yet-Another-Resource-Negotiator (YARN). In this new design, the responsibilities of the JobTracker are split between two components. A new component called Resource-Manager takes charge of resource assignment. The management of each individual

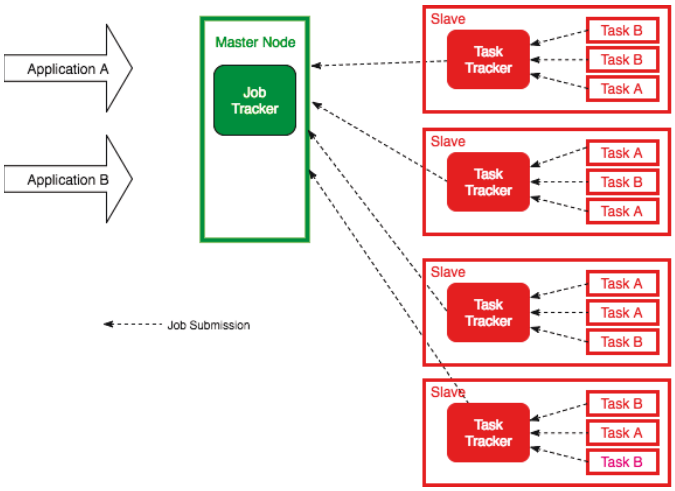

Figure 4.1: Original implementation of MapReduce in Hadoop. Taken from [2].

application is a responsibility of an ApplicationMaster that keeps track of every container working in the same application. [See figure 4.2] The resources of every node are managed at a node level by the NodeManager that continually updates the ResourceManager with the status of used and unused resources.

YARN distributes the resources using the notion of a container; the container is a representation of the allocated resources like CPU, memory, disk and network access. The ResourceManager is responsible for allocating containers to applications. Every container is assigned a cluster-wide unique identifier; the containers have detailed information about resource allocation.

Containers need to be assigned to an application by a Resource Scheduler. YARN has a modular architecture that allows the construction of different implementations of Resource Schedulers to implement different strategies regarding prioritization. The Resource Scheduler used by default in YARN is the Capacity Scheduler that is designed to run Hadoop applications as a shared, multi-tenant cluster in an operator-friendly manner, while maximizing the throughput and the utilization of the cluster. There are more options that can be used by making changes to the configuration files.

The new implementation of Hadoop based on YARN was designed to be an open, complex and multi-layered ecosystem. This ecosystem is based first on HDFS used as the base for storage and sitting on top of YARN which deals with the logic and computation. These two components can be used to do MapReduce as it was capable before, but the changes in its design also make it possible to do things beyond the MapReduce sphere such as using Spark and Tez [49]. The advantages offered by

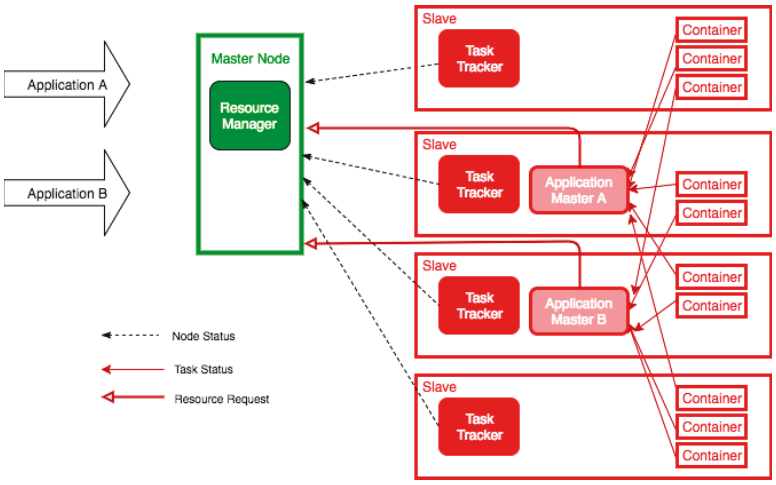

Figure 4.2: YARN MapReduce [2].

YARN are encouraging developers and researchers to build new applications on top of YARN and even on top of applications like Spark that works directly with YARN.

4.2.1.1 YARN Execution Overview

A YARN execution begins when a client starts the process by contacting a Resource Manager, requesting a new application; this means a new Application Master is created. This Application Master should be running in one of the cluster nodes. The Resource Manager surveys the current availability of resources in the nodes and decides to create the Application Master in a node that has the required availability of resources. If the application is deemed small, then it runs locally in the node; however, if the application size requires additional processing, then the Application Manager calculates the number of required additional containers and sends a request to the Resource Manager; those new containers will likely be located in different nodes. In this research using YARN to run MapReduce was one of the means of testing YARN execution. Hence it is relevant to review in detail how a MapReduce application will run using YARN.

The YARN processes are controlled by three components: Resource Manager, Application Masters, and Node Managers. Application Masters direct requests for resources to the Resource Manager in order to obtain them. The Node Managers reside one in each node and are responsible for managing the resources available in the local nodes following indications given by the Resource Manager. The Resource

Manager has a complex set of responsibilities distributed in multiple components. The following section will go through the most relevant parts of this research work which deal with the Application Master:

4.2.1.2 Application Master Service

The Application Master Service is in charge of handling responses to remote procedure calls (RPC) from all the Application Masters. The Application Master Service is responsible for a wide variety of tasks like registering and deregistering new Application Masters, managing container allocation and deallocation requests from all running Application Masters, forwarding the containers to the Scheduler and processing termination requests from a finishing Application Master.

The Application Master Liveliness Monitor works in conjunction with the Application Master Service to manage all the Application Masters and their statuses. It monitors each Application Master and the corresponding last heartbeat time. If an Application Master does not send a heartbeat within a specific configurable time, then it is marked as dead and is expired by the Resource Manager; all the containers assigned to the now dead Application Master are also marked as dead. After that the Resource Manager schedules the same Application Master to run on a new container; this restarting part is configurable by the clusters administrator.

4.2.1.3 Application Master

The Application Master is responsible for the management of a single job. The Application Master is the first container to be assigned to a job; logically there will be more than one Application Master running if multiple jobs are running. When an application is initiated, the Resource Manager calls the Application Master launcher method; it, in turn, requests a container for the Application Master process. When the container is obtained, the Application Master process is initiated, and the information about/regarding the newly created Application Master is stored for future operations. After that, the Application Master informs the Resource Manager of its creation and the supplied information includes assigned host ports, job status and job history tracking URLs.

While the application is being executed, an Application Master sends regular heartbeats to inform the Resource Manager of its condition; this is implemented to provide fault resistance functionality. The Application Master requests a number of

containers from the Resource Master; this request specifies how much memory and how many virtual cores must be assigned to every container. Those values remain the same during the execution of the application and come from the configuration files. Another part of the request that can be specified by the Application Master is for a container to be created according to the location of a part of the processed data. The Resource Manager sends in response a listing of newly allocated containers, completed containers and the current status of available resources. It is important to notice that each request includes all the containers that are going to be needed to finish the actual job; this is because, in most cases, the returned set of containers will only fulfill in part the needs of the job.

The newly allocated containers are processed by an Application Master executing the Container Launch Context method; this specifies the container ID and the local resources required by the executable. After this, a container notice is submitted to a Start-Container Request method which starts the container. After all the containers are finished and the job has been completed, the Application Master sends a Finish Application Master request to the Resource Manager, which deregisters the Application Master and begins the completion process.

4.2.1.4 YARN Schedulers

A Yarn Scheduler is responsible for allocating resources to applications following some specific policies. Hadoop is built with three ready-to-use YARN schedulers: Capacity, Fair, and FIFO scheduler. The Capacity Scheduler is designed to be able to read a configuration file with a hierarchical definition of groups and users, with percentages of cluster capacity assigned to them. The Fair Scheduler is built to share the resources of the cluster in an equal way with all the jobs being executed at any given time; this means that the cluster will be fully utilized as long there are jobs running. The FIFO scheduler is the most basic of the three because it just makes a queue of all the requests for resources and fulfills them in a first come first serve manner; this can be problematic for a multiuser cluster where small jobs can be delayed for a long period because they are in a long queue. It is important to note that the API provided by Hadoop makes it possible for a user to make a custom Scheduler that fits the specific needs to achieve an objective.

4.2.1.5 Tunable YARN Parameters

YARN utilizes hundreds of parameters, and a small number of them have an extremely high impact on performance. Most YARN tuning is done using a manual iterative strategy; the information about carrying tuning is scattered among multiple sources such as white papers, websites, and blogs. The most important parameters to tune in YARN are [50] [51]:

- yarn.nodemanager.resource.memory-mb
- yarn.nodemanager.resource.cpu-vcores
- yarn.scheduler.maximum-allocation-mb
- yarn.scheduler.minimum-allocation-mb
- yarn.scheduler.maximum-allocation-vcores
- yarn.scheduler.minimum-allocation-vcores

4.2.2 KERMIT Auto Tuning

KERMIT is the solution that our group developed to offer automatic tuning for a YARN-based application; it was built using Java and currently operates with YARN, being used by Hadoop and Spark. The overall working of KERMIT will next be briefly summarized; after this will follow a more detailed description of KERMIT:

- KERMIT was built to be able to tune the memory and the virtual cores assigned to the containers; these values can be changed on the fly to the new containers being created.
- The containers completion time information is obtained inside YARN, and that information is used as a way of measuring the performance of the application. A shorter time for running the containers is an indicator of better values for the configuration.
- The performance evaluation of the configuration parameters is performed by KERMIT; this evaluation is used to decide what parameter values should be used to achieve the best performance.

- KERMIT was built to be agnostic about application and container types; the tuning process does not take into account that kind of information and only container completion time is used.

In order to associate a cost for a parameter configuration in a given period of time, the group used the average container duration; this cost measure is used in KERMIT as the way to evaluate performance.

4.2.2.1 Concept of Waves

It is vital for following KERMIT design to have a clear understanding of two kinds of waves inside YARN.

- YARN wave: A YARN wave is created on every YARN heartbeat. All the containers of a YARN wave are started at the same time and have the same parameters for memory and virtual cores.
- Autotuner wave: A concept created along with the development of KERMIT. An Autotuner wave is created after every specific period of time; this time will normally include more than one heartbeat. Hence Autotuner waves contain the containers coming from several YARN waves. The purpose of an Autotuner wave is to be able to evaluate the performance or cost of a specific parameter configuration. It is also worth noticing that the duration of Autotuner waves is configurable inside KERMIT; this can be adjusted to match different scenarios.

These two different types of waves can be seen in figure 4.3.

4.2.2.2 KERMIT Placement within YARN

The KERMIT Autotuner works inside the Application Master Service. Resource requests are intercepted and read by the Autotuner; those requests, along with the new container information collected by the Application Master Service, are sent to a tuning algorithm. The cost statistics are analyzed by the tuning algorithm, and the configuration values controlling KERMIT Autotuner will be changed if necessary. After the parameters to be used have been determined, a new Autotuner wave is created. This newly created wave will use the selected parameters for a user-defined period of time. The Autotuner calculates the number of containers that can be created

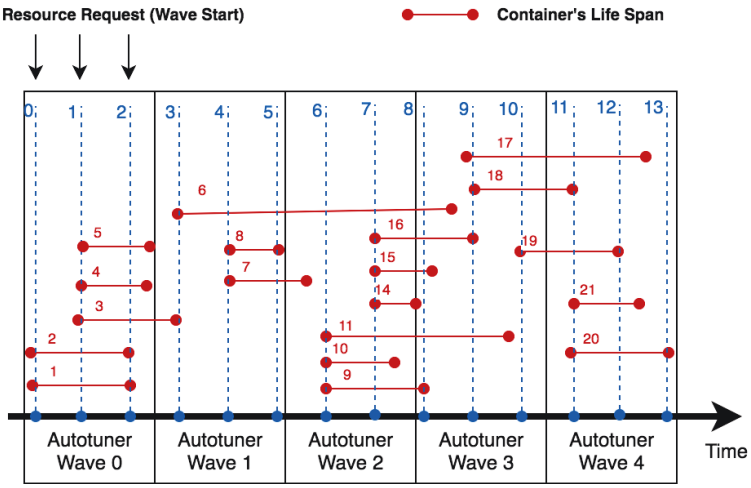

Figure 4.3: Autotuner waves.Taken from [2].

with the selected parameters and sends that number to the Application Master Service that then proceeds to the allocation of the required containers.

This process of reviewing the container statistics and creating new Autotuner waves based on that analysis goes on during the entire duration of the job. When all job tasks are completed, the Application Master Service unregisters the Application Master and the detailed information of container times is written into the log; this information was used by the KERMIT team to do the initial analysis and to verify that the tuning algorithms were reacting as expected. It is worth pointing out that the KERMIT design is built in a modular way that allows the extension and creation of tuning algorithms that will be totally independent of the original KERMIT design. [See Figure 4.4]

4.2.3 Architecture

Along with the execution of the Application Master Service, an Autotuner object is always created under KERMIT. The Autotuner holds some control variables used in the tuning and statistics process, but it also maintains a reference to an object of the type of an Algo, and a Wave statistics type of object called WaveMetrics.

Algo is an abstract Java class that must be extended by every tuning algorithm made to work inside KERMIT. The main method of the Algo class is the toTune method that receives the parameters used in the last wave and analyzes the statistics

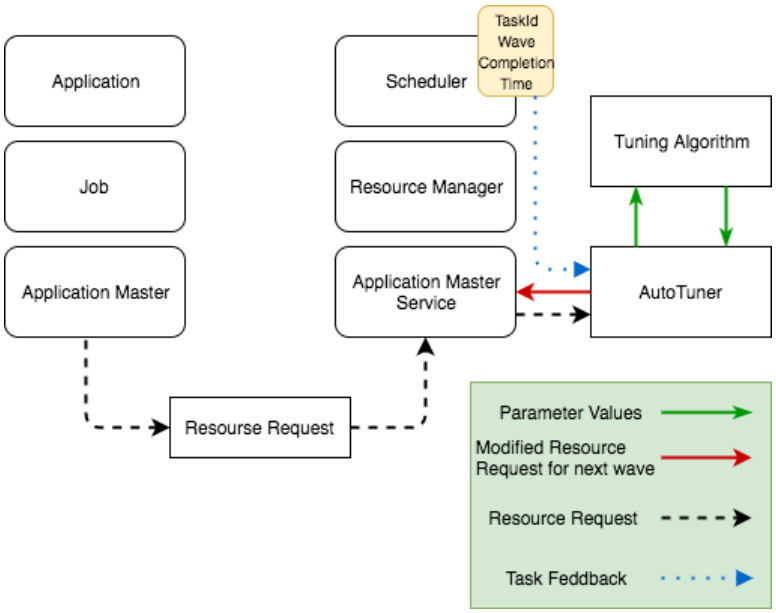

Figure 4.4: Autotuner placement in YARN.Taken from [2].

to decide which values to use next. The details about deciding which configuration will yield the best performance are completely up to the developer in charge of implementing the Algo subclass. KERMIT was originally implemented with three Algo implementations: HCAlgo, which is a Hill Climbing implementation of Algo, and two implementations of the Explorer algorithm the three are used to explore the search space to try to find the optimum value for a lab test.

4.3 Past Breakthroughs

In this section the breakthroughs that were achieved by this research team before this research work began will be reviewed. Breakthroughs of the team that consisted of Mikhail Genkin, Maria Pospelova, and Yabing Chen were as follows:

- On the fly tuning: The biggest achievement in this project was to have a workable manner to change the configuration values in YARN without the need of restarting the application cluster; this functionality is not available in the YARN code and, because of that lack of official support, there were no assurances that this could actually be done without massive changes to YARN. The team thus created some additional classes that were embedded into the YARN libraries.

- Tuning Hadoop/YARN: The team was able to run tuning algorithms seeking to optimize the performance of Hadoop/YARN applications. YARN is not only used in Hadoop. Initial tests were successful on Hadoop and it is noted the additional frameworks that use YARN can theoretically be tuned.

- KERMIT Framework: KERMIT was designed and built to become the foundation for subsequent projects that will try to tune YARN. The design was done in such a way that the creation and testing of a new tuning algorithm was a straightforward task, because the design of KERMIT allows having multiple pluggable components that can be selected to be in charge of the tuning process.

4.4 Past Points for Improvement

In this section specific issues will be identified that were points for improvement when the author of this research work started working in the KERMIT project team. Reviewed will be how they were affecting KERMIT, what the known possible causes for the issues were and what the plans were to work on them.

- The KERMIT Autotuner was built, and it was successfully changing tuning variables and analyzing statistics, but the overhead of tuning was still too significant to see real performance gains. Thus there was no performance improvement so far.

- Continuous tuning: The tuning algorithms that were developed by the research team were built to be continually searching to improve the performance of the cluster. This permanent search for optimum values leads to a considerable time overhead for the cluster; a new strategy was needed to be able to perform parameter tuning only until a reasonable efficiency threshold was achieved.

- Ability to tune another YARN framework: Hadoop/YARN was successfully tested using KERMIT; however, one of the YARN's strong points is that it is not tied only to Hadoop. It would be interesting to see how KERMIT performs outside of Hadoop in a framework that uses YARN only.

- Logging of KERMIT operation: KERMIT had logging that helped enable the team to know whether it was executed correctly or not. This information was

useful for the team but was in a format only readable by humans and it was hard to process statistics from it.

Chapter 5

KERMIT Enhancements

5.1 Introduction

This chapter will review the contributions made to the KERMIT project by the author. Some of the information in this chapter has been presented in HPCC 2016 [5].

5.2 Analysis of KERMIT Performance

Using KERMIT to tune YARN parameters proved to be feasible; the parameters were being changed dynamically from within YARN without having to change configuration files or having to restart the cluster. To be able to analyze in depth the effect KERMIT was having inside YARN, additional tools were needed.

KERMIT tuning is based on analyzing the number of containers that are finished inside a tuning wave; the tuning algorithm will analyze the number of containers finished during past waves and will decide which values to use for the next wave. With KERMIT being so dependent upon those two values, it motivated the team to look for a way to enable the extraction of that information in a streamlined manner.

The original version of KERMIT already had some logging that could be used to obtain an idea of how the cluster was working and how the tuning algorithm was analyzing performance metrics to tell KERMIT to use a particular parameter value. That information was written in the log that was meant to be read by a human only who would draw conclusions from that. An analysis like that was feasible when there were only a handful of runs to analyze, but, as the tests progressed, it was clear that the container and wave information would be more useful if the team could take that information and feed it to special tools like spreadsheets and custom scripts to process

and analyze it.

KERMIT was using the same log file as was used by the YARN Resource Manager; this had some positive and some negative points. The good part regarding using that file was that the team could see where in the context of the Resource Manager the KERMIT operations were being executed; this led to a better understanding of YARN structure. The negative part of using that file was that the Resource Manager logs a great deal of detailed information that, most of the time, was not very relevant for the team and they had to browse very large text files to see only a few log entries coming from KERMIT. The team decided that it was important to modify the way information was being obtained from KERMIT in order to make decisions on how to improve the tuning.

5.2.1 Enhanced Logging Infrastructure

The changes performed to improve the logging aspect of KERMIT were:

A new custom class called KermitLogger was created in the code; the purpose of this class was to keep KERMIT logging entries inside the YARN Resource Manager log, but to also have a separate KERMIT log file that would only contain information sent from KERMIT. This allowed the team to have the best of the two worlds in the sense that it was still possible to see the KERMIT entries inside the context of the larger Resource Manager log, but it was also possible to analyze the far more manageable KERMIT log that would be the main source of information for the team.

KERMIT was modified to include additional methods specially designed to be executed at the end of a job. The data printed in the log were the wave and container statistics and this data can be seen in Tables 5.1 and 5.2; that data is printed without a particular format, and it was intended to be picked up by two custom Java programs built by the team.

Table 5.1: WaveStatToCsv example.

Wave Id	Memory	VCores	AverTime	StDev	Last20Avg
0	1024	1	56297	32152.84	71064.23
1	3072	1	74478	47417.72505	110976
2	5120	1	109812	68912.74168	124208.8125
3	6144	1	72579	33825.44142	79797.4918

Table 5.2: WaveMapToCsv example.

ContainerId	WaveId	Minutes	FinishWave	StartTime
2530_0001_01_000038	0	6.15	3	14:36:56
2530_0001_01_000037	0	5.81875	2	14:36:47
2530_0001_01_000185	1	0.685383333	1	14:38:19
2530_0001_01_000184	1	1.303833333	1	14:38:16

Those programs are:

- WaveStatToCsv: This program reads the summary data for each tuning wave; it outputs the same statistics used by the tuning algorithms, such as average container duration time during each tuning wave (AverTime), average container duration time during the last 20 seconds of the wave (Last20Avg), and the standard deviation of container duration time during the wave (StDev). The output of this program is a comma-separated values file (CSV), with a summary for each tuning wave.

- WaveMapToCsv: This program reads the raw data for each container used in the job; the start and end times are the focus here. The output of this program is a CSV file with the information regarding every container.

Having these new tools at the disposal of the research team made it easier to analyze the tuning process. The addition of these information extraction tools paved the way for the team to find improvements in the tuning process.

5.3 Not Continuous Tuning Approach

One of the major problems the team faced was that the use of KERMIT infrastructure to tune the performance of YARN was not sufficient to improve the overall performance of a job that was configured using a rule of thumb (RoT) configuration. This was one of the most important aspects for the team to try to improve because it was important to be able to show that using our approach could result in a real improvement in execution times.

The Hill Climbing (HC) algorithm initially built for KERMIT was found to take too long to find good configuration values. The gains obtained with tuning were overshadowed by the overhead of exploring many configurations that resulted in jobs taking much longer to finish with the use of AutoTuning. The way that HC works forces the process to go through many tuning waves searching for optimal values; the search space in the case of the YARN jobs and the selected tunables was neither too complex nor too big. The nature of the search space and the complexity of HC made the team reconsider whether or not it was the best choice for the task at hand. It was also clear that HC was also very sensitive to statistical noise; this resulted in choosing values that were not optimal and taking longer to finish tuning.

The Explorer algorithm developed by the research team was built to continuously iterate through the parameters in a search for the best configuration at any given time. This was originally intended to test a range of parameters to be able to tell which configuration was achieving results at the best cost.

5.3.1 Explorer Algorithm V2

The Explorer algorithm was then taken as the basis to create a new tuning algorithm that could do fairly good tuning, but with less overhead than the implementation of HC (Hill Climbing). The same structure as the original Explorer would be the basis, but additional constraints would be set to make it an efficient tuning algorithm.

The original Explorer had predefined lists of explorable values for the tunables. The difference in the new version is that these lists would not be explored for cost in an unlimited number of tuning waves; rather, there would be a limit where the results for all the values would be compared and used to decide which value was showing the better cost. This is a significant change from HC where the overhead in doing hill climbing can be prohibitively expensive in terms of the number of tuning waves.

5.3.1.1 Explorer Algorithm Execution Overview

Explorer was modified to iterate two times through a list of values for container memory or container virtual cores; after that iteration, statistics are gathered and processed to decide the tunable configuration with the best cost. This process was named Global Search because it is the biggest search that is going to be performed by the Explorer tuning algorithm.

After finishing the Global Search phase, Explorer enters into an Observe state that will continue to use the configuration deemed to have the best cost found during the Global Search. By setting a threshold for cost increase for the selected values assigned to the YARN configuration values, a state is set in place to avoid unnecessary tuning attempts. This means that the Observe state can only be interrupted if the cost threshold is surpassed for a given number of tuning waves (The default configuration is set to three waves).

When the Observe state is interrupted by a rise in the cost of the performance, the Explorer algorithm transitions to a new phase called Local Search. This phase is designed to do a subtle adjustment that will be likely to improve performance but will not create the large overhead associated with a Global Search. The Local Search is optimized to do a search looking for a better cost only in the two nearest values to the actual configuration. This reduced search space will result in less overhead than a Global Search, but with a good chance of finding a sweeter spot for the configuration values. When this shorter Local Search is finished, then a choice is made as to which configuration is giving the best results. This configuration is selected as the active configuration, and the algorithm goes back to an Observe state that will continue to use that configuration until the same conditions are met to transition to perform another Local Search.

Since this new tuning process is totally different from the original Explorer, the team opted to name it Explorer v2; the overall flow of this tuning algorithm is illustrated in the Figure 5.1.

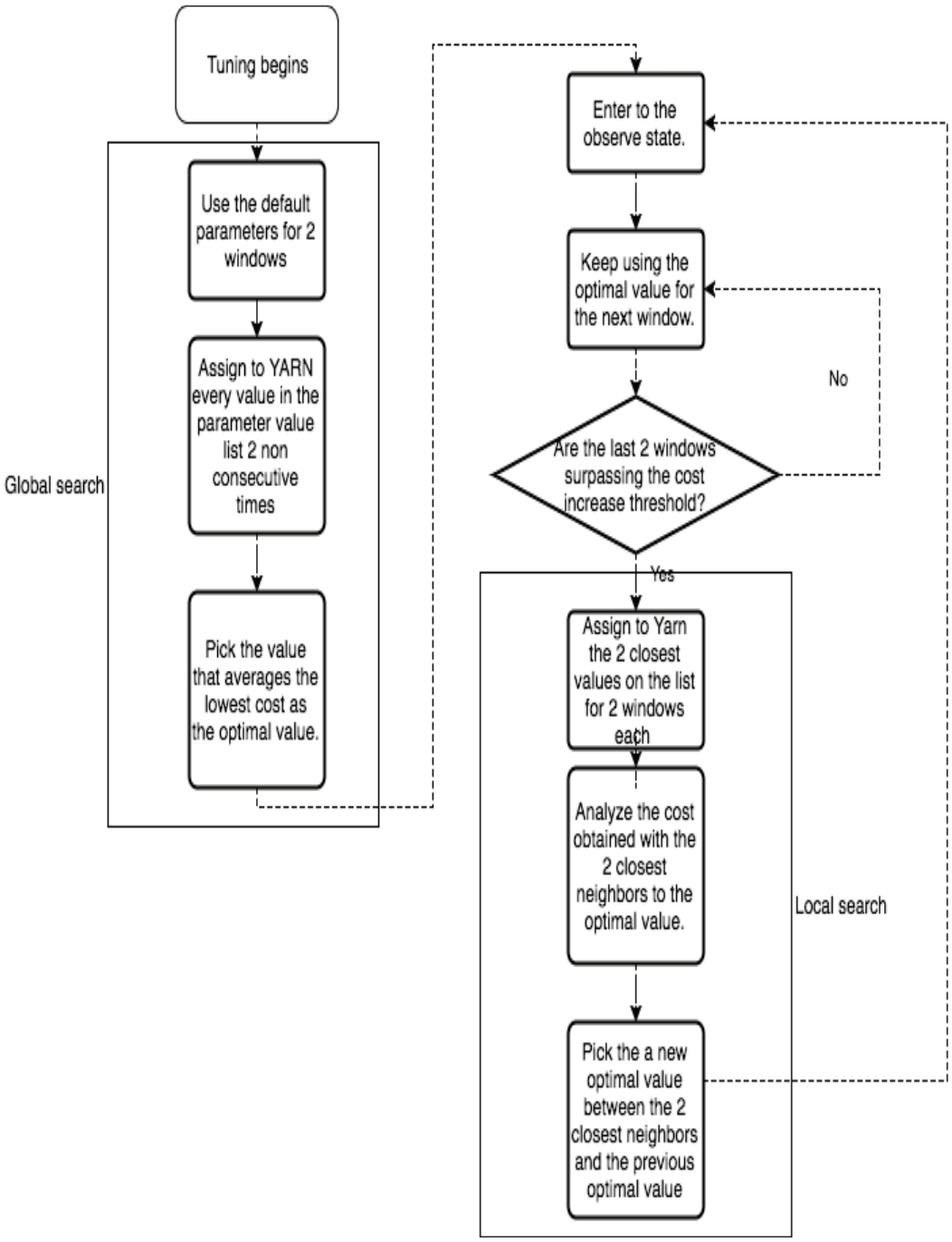

Figure 5.1: Explorer algorithm version 2 flow chart.

Chapter 6

Data Analysis

6.1 Introduction

To test KERMIT's performance with the enhancements described in Chapter 6, YARN jobs were run in a cluster. In this chapter, the experiments that were performed using these enhancements will be reviewed. This chapter information has been presented in HPCC 2016 [5].

6.2 Cluster Setup

All the tests were run on a four-node cluster. The distribution of the YARN cluster was: one node was used for management and three nodes were configured as data nodes. All the nodes were using the Linux distribution Ubuntu operating system version 14.04 and YARN 2.6.0; no virtualization was utilized in this setup thus this cluster can be considered to be using an all-metal (or bare-metal) configuration. Every data node was equipped with one SSD (solid state drive), which was used to host the operating system and the YARN/Hadoop stack installation, and four data disks each with 1.8 TB of memory. Each data node was also equipped with 32 GB of RAM and 1 Intel i7 CPU with four physical cores running at 1600 MHz. Half of the virtual CPUs visible at the operating system level were disabled via an operating system command to reduce noise caused by I/O wait.

6.3 Autotuning YARN in Hadoop and Spark

One of the advantages of doing tuning in YARN is that YARN is used in many data processing applications. YARN was first designed as a way to increase reliability in Hadoop; because of this it was natural to try to tune YARN in a Hadoop context. Using KERMIT to run inside a Hadoop YARN installation was one of the first breakthroughs achieved by this research team; however, it was also important to be able to probe the concept that KERMIT can work in different YARN applications. Spark was selected as another good candidate to be tuned by using KERMIT. Hadoop and Spark have some differences that will be addressed in the coming sections where we will describe how the tests were performed and the data was obtained.

6.4 Benchmarks

YARN/Hadoop benchmarks can provide information on what is the level of performance for a cluster [52]. In order to measure the performance gains obtained in YARN jobs by the use of KERMIT autotuning selecting some specific benchmarks was necessary. Since Hadoop and Spark do not operate the same way, the team had to use separate benchmarks.

6.4.1 Hadoop Benchmarks

In the case of Hadoop, the team opted to choose two benchmarks as the primary workloads for the performance comparisons:

- Terasort: This is a common Hadoop benchmarking application. It is commonly included in Hadoop builds, and it was chosen because it is known by virtually every member of the Hadoop user community. This benchmark consists of generating a file that is filled with a particular quantity of random numbers (the name for this part of the process is TeraGen); after that this file is then sorted by the cluster using a MapReduce algorithm (this is called the TeraSort part). The sorted result can be validated to verify that the process was successful (this part of the process is called TeraValidate). For the particular case of this benchmark, the only process that was executed and monitored by using KERMIT autotuning was TeraSort; this decision was taken to be able to use a

very isolated job as a simple means to test KERMIT and thus making the runs less similar to TPCx-HS [53].

- TPC Express Benchmark^{TM}HS (TPCx-HS): This benchmark is a recent industry standard made by the Transaction Processing Performance Council (TPC). It was inspired by the TeraSort/TeraGen/TeraValidate utilities that have been commonly used by big data practitioners for several years. TPCx-HS requires that the generate, sort and validate jobs sequence is executed sequentially without interruptions [54]. It specifically disallows manual tuning between the benchmark stages, but it explicitly allows for automatic tuning. The processes that forms part of TPCx-HS are HS-Gen, HS-Sort, and HS-Validate; these processes are in a broad sense similar to the set of processes that compose TeraSort, but they cannot be treated as equal because their implementations are not identical. Since HS-Gen, HS-Sort and HS-Validate perform very different types of processing, it is hard to find a good combination of tuning parameters that work well for all three of these stages manually. For these reasons, TPCx-HS was judged to be a very good benchmark for our study.

6.4.2 Spark Benchmarks

To select Spark benchmarks, a special criterion was needed because the containers function differently compared to those in Hadoop. In Hadoop it is very usual to have containers being created and disposed all along the duration of a job; Spark containers are created as executors at the beginning of the execution and all of the tasks that are part of the job run in those containers. The Spark executors will remain unchanged until the job is finished.

Because of this situation there was a need to use a tuning method that would allow the team try to do autotuning under these conditions. Thus for a useful online automatic tuning test, the team needed a multi-job workload. Spark Multiuser Benchmark (SMB) executes multiple identical Spark TeraSort jobs concurrently and reports throughput and job duration statistics for these jobs [55]. For this research, the team used a 2 GB data scale for the Spark TeraSort implementation and 10 concurrent users for all tests. Our analysis was focused on the throughput metric. The number of Spark executors was left at the default value. Spark 1.6.1 was used for these experiments.

6.5 Evaluation Methodology

The evaluation methodology focused on comparing the end-to-end job response-time achieved with KERMIT automatic tuning with two baselines:

- Basic tuning baseline: The reason for using a Basic tuning baseline was to make sure our approach was yielding a clear, measurable performance improvement relative to simple manual tuning typically performed by big data practitioners. The basic tuning baseline did not use strictly out-of-the box (OOB) settings. Instead, it was based on a shallow-tuning approach typically used by field practitioners. To achieve the basic tuning baseline, the OOB Hadoop or Spark configuration was taken as the starting point. Then the yarn.nodemanager.resource.cpu-vcores parameter in the yarn-site.xml, file was set to equal to the total number of CPUs shown by the operating system on each of the cluster nodes. The yarn.nodemanager.resource.memory-mb was set to equal to the total amount of memory on each data node. In the mapred-site.xml the only parameter that was tuned was the mapred.child.java.opts setting which was modified to increase the maximum JVM heap size setting from the default value of 200 MB to 890 MB. This was done to remove the possibility of a memory bottleneck that could impact the performance of both the baseline and the auto tuning runs, and to ensure that both the baseline and the auto tuning runs had the same JVM heap size and only automatically tuned YARN-parameters differed. This tuning approach is similar to the type of tuning commonly performed by field practitioners.

- Best possible tuning (exhaustive search of parameter space): To achieve the best possible tuning baseline Hadoop or Spark settings that were being intercepted and automatically tuned on the YARN side by KERMIT were searched exhaustively by using Explorer V1, not with the intent to tune but only as a tool to explore the search space and to analyze the data manually. The settings that were tuned on the Hadoop side are (1) mapreduce.map.memory.mb; (2) mapreduce.map.cpu.vcores; (3) mapreduce.reduce.memory.mb; (4) mapreduce.reduce.cpu.vcores. Spark parameters tuned for this study were: (1) spark.executor.memory; (2) spark.executor.cores.

  For the CPU settings values were tested between 1 and 8 virtual cores. For the memory settings, the values were tested between 640 and 3200 MB for

Hadoop and between 1024 MB and 12288 MB for Spark. The benchmarks used were TeraSort, TPCx-HS and SMB. For TPCx-HS the procedure defined in the TPCx-HS specification was strictly followed. At least 5 data points were collected for each combination of settings to ensure the results were repeatable, and the average end-to-end duration or throughput was calculated. Results were plotted to show the shape of the search space. The best combination of settings and end-to-end benchmark duration were used as the best possible tuning baseline for the automatic tuning comparison. For TeraSort and TPCx-HS, this procedure was repeated for several data scales - 300 GB, 500 GB, 800 GB, 1 TB and 2 TB. It is logical to assume that KERMIT will not reach the same level of performance as the best possible tuning because the algorithm would have to explore the parameters to get to the most efficient configuration; there is an inherent overhead in doing this. It must be noticed that performing this best possible tuning was a long process that involved running the jobs repeatedly and then analyzing the results; this is the kind of tuning the team wanted to avoid by using KERMIT.

As was stated in earlier chapters, KERMIT automatic tuning can use different tuning algorithms; for these experiments the best option was Explorer V2 because it offers a low overhead regarding waves and a good tuning accuracy for the search space. The reader can assume that Explorer V2 was used for all the tuning runs of this experiment; for the sake of readability Explorer V2 may be referred as Explorer.

Comparison job runs done using KERMIT on-line automatic tuning were performed at the same data scale. Several automatic tuning runs were performed to ensure consistency of the results. The comparison focused on three aspects:

1. Assessing the level of improvement compared to the basic tuning baseline.

2. Assessing the level of improvement compared to the best possible tuning baseline.

3. Assessing how the first two results change with an increasing scale of data, determine if an increase of data size impacts positively or negatively the tuning performance.

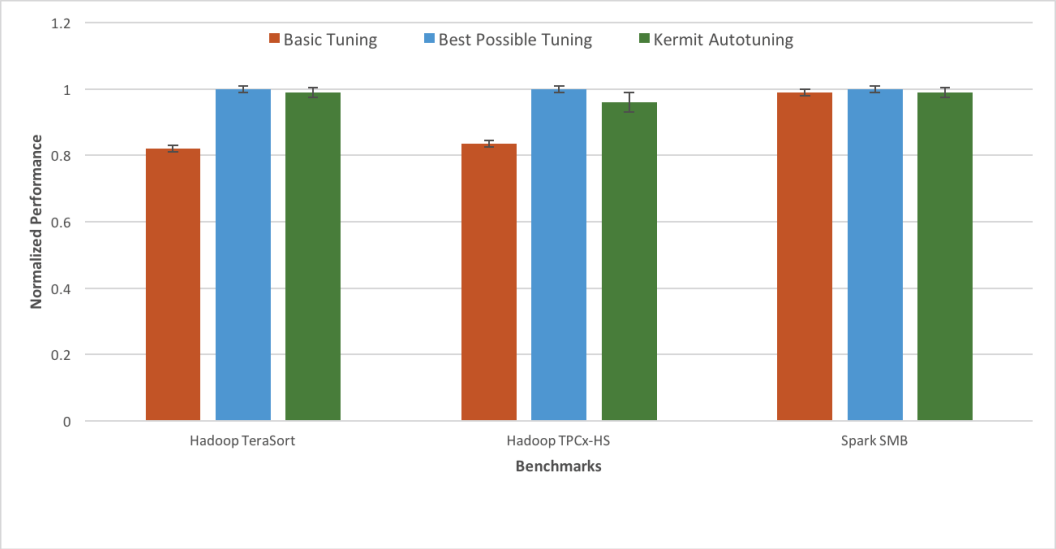

Figure 6.1: Normalized performance for Hadoop and Spark benchmarks.

6.6 Results

Figure 6.1 summarizes the normalized performance achieved by KERMIT for Hadoop and Spark workloads including error bars that show the observed standard deviation. The best possible tuning result for each benchmark was set equal to 1. For all benchmarks, the KERMIT automatic tuning result reached levels that can be considered to be very close to the best possible tuning result. In the case of Hadoop benchmarks TeraSort and TPCx-HS, the best possible tuning result and the KERMIT automatic tuning were significantly better than the basic tuning result. KERMIT automatic tuning result achieved an optimal level of performance for the Spark benchmark SMB. This can be considered a coincidence; the default container size of 1024 MB is optimal for our cluster and data scale, resulting in optimal basic tuning. In general, this is very unlikely, in particular for larger clusters. Currently, the team is working on running tests on a bigger cluster with significantly more CPU and memory resources; however that process is still ongoing, and it has not yet yielded data to be presented for the time being. In the future work chapter,Chapter 8- Future Work, this situation will be elaborated upon.

The Figures 6.2, 6.3 and 6.4 illustrate the shape of the search space for TeraSort, TPCx-HS, and Spark SMB, respectively; these values were obtained by doing manual tuning experiments. The objective of these experiments was to be able to determine

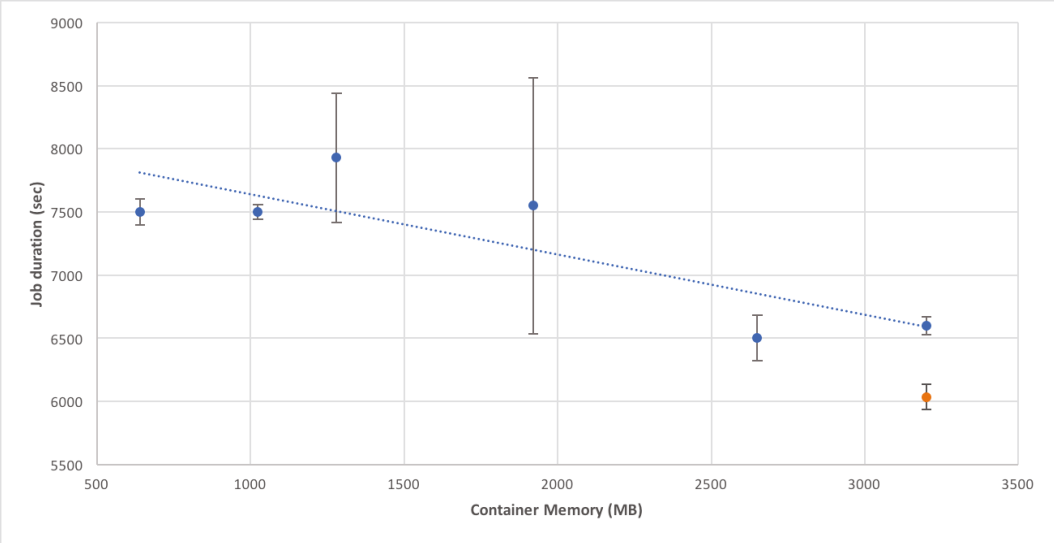

Figure 6.2: Data for best possible tuning for TeraSort at 500 GB data scale.

which was the best possible tuning baseline for container memory. In all the cases the data points shown are averages of data points collected at those memory settings. The y-axis error bars show the standard deviation of the data. The KERMIT automatic tuning result for the matching data scale is shown as the orange marker with error bars that denote the standard deviation of the autotuning results. The position of the KERMIT autotuning data points on the x-axis indicates the container memory value that the Explorer algorithm settled at. For Hadoop benchmarks, the y-axis records the job duration (response-time). In this case the lower values are better; a lower response time is indicative of higher performance. For Spark SMB, the y-axis records show throughput as measured in jobs per hour. In this case the higher values are desirable; higher throughput is indicative of higher performance.

In the three previously discussed cases, the search space has a global optimum (within constraints discussed in the Evaluation Methodology Section 6.5) as well as local maxima and minima. For all the cases the team's Explorer V2 algorithm was able to find the global optimum (maximum for SMB and minimum for TeraSort and TPCx-HS), and deliver a configuration that was statistically very close to this optimum.

To obtain a more detailed view of how the Explorer algorithm operates, one can review the Figures 6.5 and 6.6. Those figures show how Explorer V2 behaved during

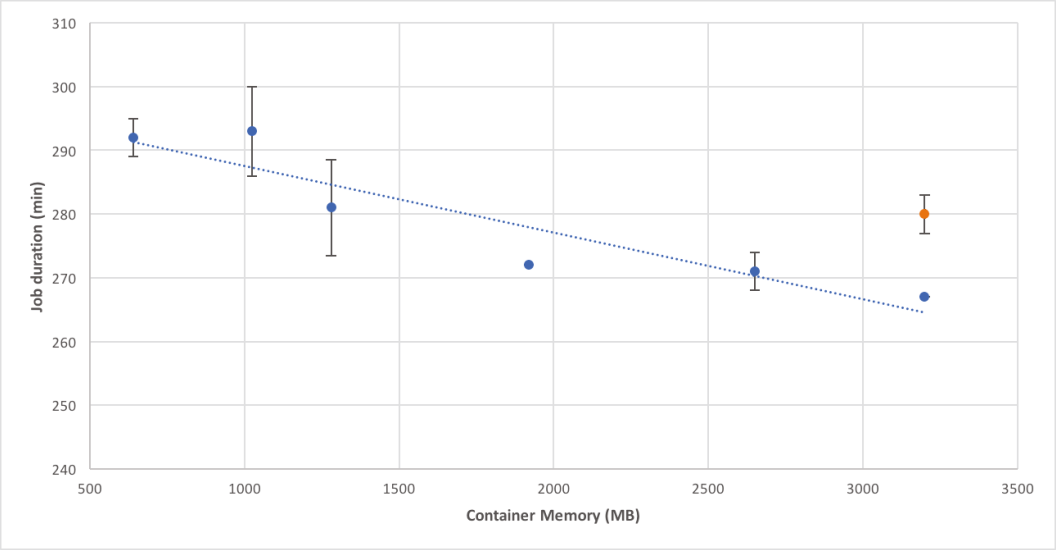

Figure 6.3: Data for best possible tuning for TPCx-HS at 500 GB data scale.

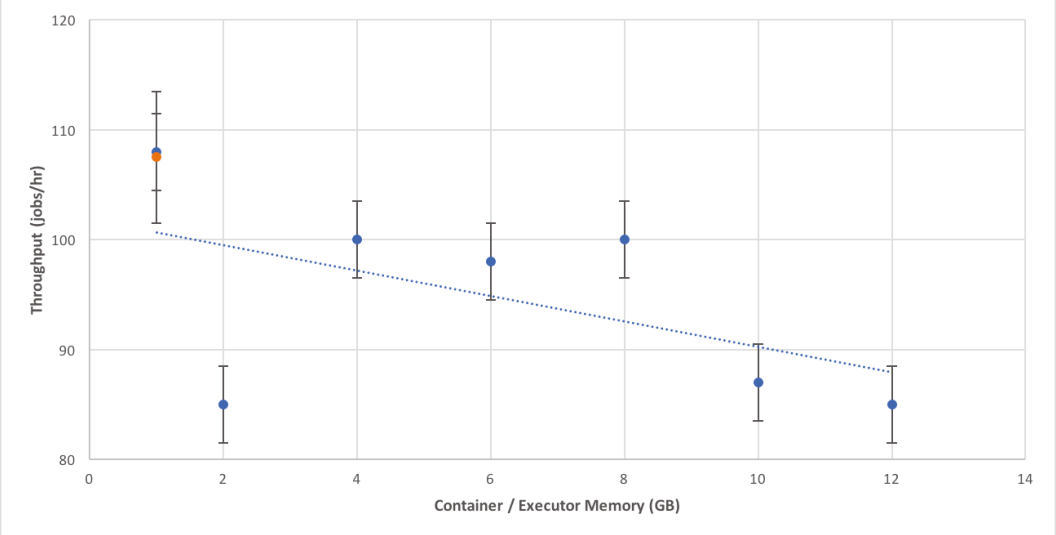

Figure 6.4: Data for best possible tuning for SMB.

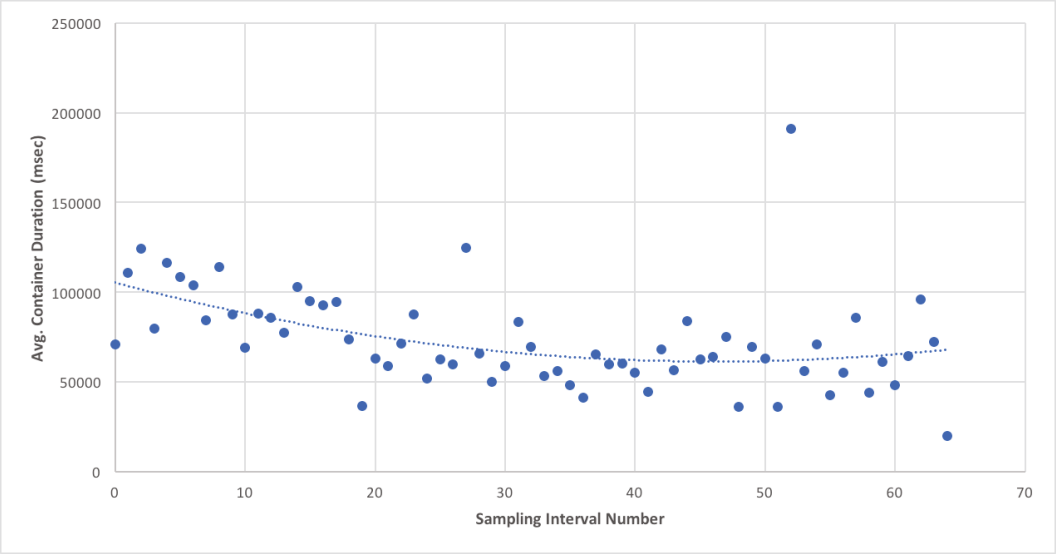

Figure 6.5: Explorer algorithm optimizing container durations for SMB workload.

SMB execution. Figure 6.5 shows container duration averages collected during all the sampling intervals of one of the SMB runs. The Figure 6.6 shows that container memory values were modified by the KERMIT during this same SMB run. Despite the fact that with SMB we were trying to optimize throughput, the Explorer V2 algorithm aimed to reduce the container duration observed during its sampling intervals. This specific situation with SMB illustrates that our KERMIT autotuning approach is not just tweaking execution to satisfy one metric, but also other metrics that were not considered at the start of the project were seeing improvement.

During the initial explore phase, Explorer iterates through all of the predefined memory values defined in the previously specified ranges. These iterations manifest as a diagonal alignment of data points in Figure 6.6. Once Explorer finds the optimal container memory configuration, it maintains this configuration for all subsequent sampling intervals until it detects a change from the steady state as previously explained in the previous Chapter 5- KERMIT Enhancements. This is clearly shown in the horizontal alignment of data points in Figure 6.6. Figure 6.5 shows that average container durations are consistently shortened after Explorer finds the optimum container memory value.

It was also relevant to try to establish the effects of data scale on the efficiency of the Explorer algorithm; these effects were investigated using the TPCx-HS bench-

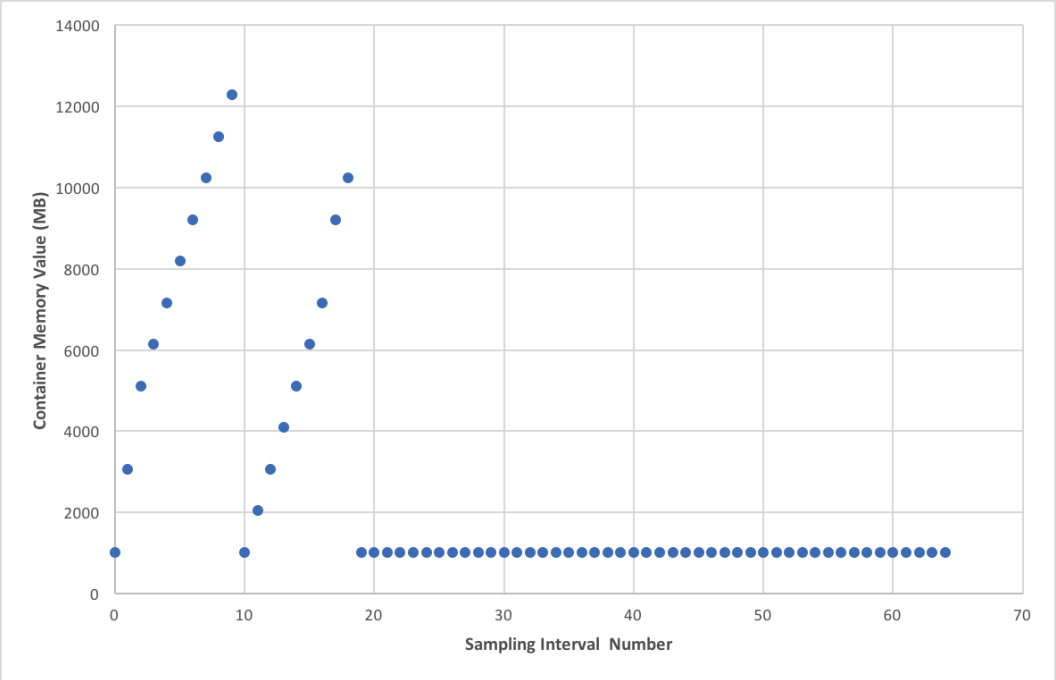

Figure 6.6: Explorer algorithm optimizing container memory allocations for SMB workload.

mark. Table 6.1 shows the data collected for TPCx-HS at different data scales ranging from 300 GB to 2 TB. Data in the column labeled Basic Tuning shows values collected for the basic tuning baseline discussed in the preceding section. Data shown in the column labeled Best Possible Tuning shows values collected for the best possible tuning baseline discussed in the previous section. Data in the column labeled KERMIT Tuning shows comparison values collected with KERMIT on-line automatic tuning. As was expected, KERMIT data points fall in between the two baselines. This was predicted by the team because KERMIT has some implicit overhead to perform the tuning; thus it is very hard to achieve better performance than the best possible tuning. It can be easily noticed that, as the scale of the data grows, the difference between KERMIT tuning and basic tuning is becoming more significant; on the other hand, the difference between KERMIT tuning and best possible tuning is becoming less significant.

Table 6.1: TPCx-HS - KERMIT automatic tuning efficiency for various data scales.

Data Scale (GB)	Basic Tuning Runtime (min)	Best Possible Tuning Runtime (min)	KERMIT Tuning Runtime (min)
300	172	159	166
500	291	266	274
800	481	438	449
1000	624	542	566
1500	977	852	865
2000	1350	1165	1179

The data points obtained when establishing the best possible tuning for TPCx-HS at 300 GB data scale are illustrated in more detail in Figure 6.3. For the experiments shown in this figure the values of mapreduce.map.cpu.vcores and mapreduce.reduce.cpu.vcores were kept at the default of 1 (experiments where these values were changed from the default value were also executed). It is interesting to note that when mapreduce.map.memory.mb and mapreduce.reduce.memory.mb were set to the default value of 1024 MB, this produced the worst TPCx-HS performance. The best TPCx-HS performance was detected when these container memory configuration values were set to 3200 MB. A similar pattern was observed at all the other data scales that were executed.

Figure 6.7 compares KERMIT on-line automatic tuning with the best possible tuning of the TPCx-HS benchmark for data scales ranging from 300 GB to 2 TB. The y-axis shows the improvement (in %) gained relative to the basic tuning baseline explained above. At small data scales around 300 GB, the performance gain with even the best possible tuning is relatively modest. However, it increases with rising data scale to achieve nearly 14% at 2 TB. Likewise, the improvement achieved with KERMIT on-line automatic tuning is relatively small at small data scales but grows to nearly 13% at 2 TB. It can be noted that KERMIT tracks the optimal tuning curve rather closely. This is because KERMIT at some point starts to use the same values as the best possible tuning and the overhead after that point tends to be very small.

The performance of KERMIT on-line automatic tuning for TPCx-HS shows a

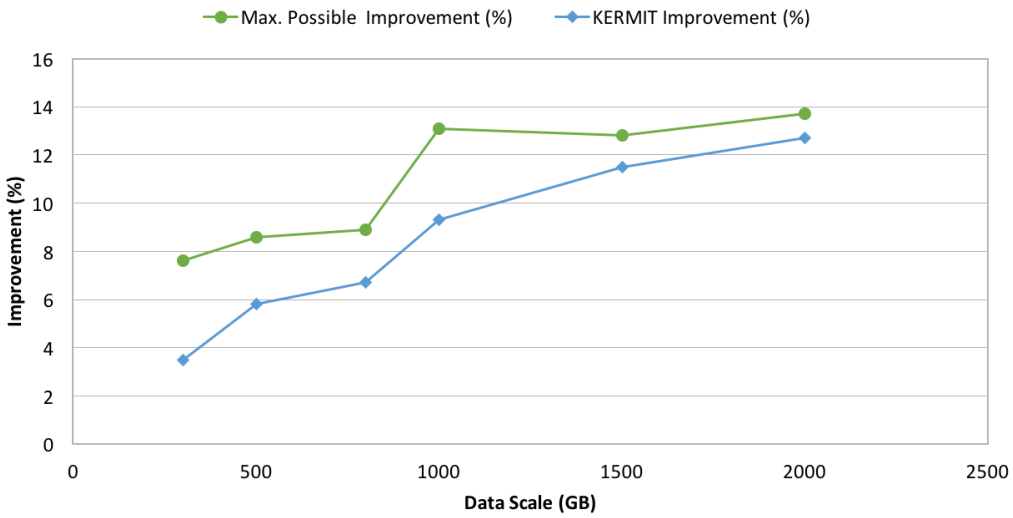

Figure 6.7: Comparison of KERMIT on-line automatic tuning with the best possible tuning for TPCx-HS at different data scales.

linear increase with growing data volume; this can be observed in Figure 6.8. It is relevant to point out that KERMIT was able to demonstrate this at much larger and more realistic data scales compared to previous efforts such as MRONLINE [25] (2TB for KERMIT vs. 100 GB for MRONLINE), and that these gains are compared to a tuned configuration (basic tuning) instead of the default YARN configuration used for MRONLINE. When projected to larger data scales; the total performance improvement obtained with automatic tuning can be predicted to continue to grow. Most actual big data setups operate at data scales larger than 5 TB. It is unlikely that the performance improvement trend will remain linear as the data scale continues to increase. However, at 5TB and above, we expect performance improvement to be between 20% and 30%.

The research team wanted to evaluate the efficiency of KERMIT tuning as the data size was increasing. To do so it was necessary to use the TPCx-HS averages previously shown in the table 6.1. The mathematical formulas used to do the calculations are illustrated in equations 6.1, 6.2 and 6.3. Those formulas were used to calculate the values in each data scale to come up with the information regarding the efficiency of KERMIT on-line automatic tuning.

$$best\ possible\ gain = basic\ tuning\ time - best\ possible\ tuning\ time \tag{6.1}$$

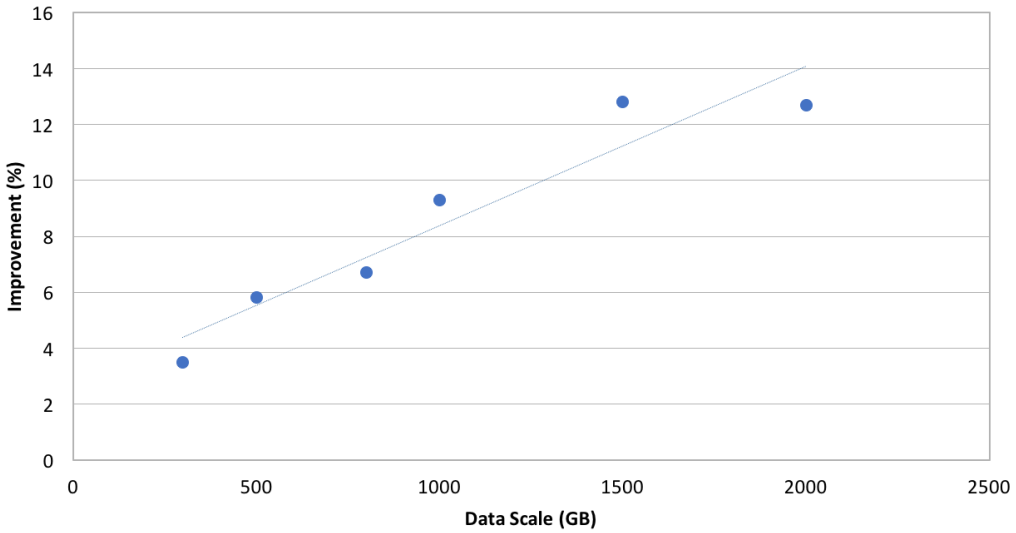

Figure 6.8: Performance of KERMIT on-line automatic tuning for TPCx-HS shows a linear increase with increasing data volume.

$$actual\ KERMIT\ gain = basic\ tuning\ time - KERMIT\ tuning\ time \tag{6.2}$$

$$KERMIT\ efficiency = \frac{actual\ KERMIT\ gain}{best\ possible\ gain} \tag{6.3}$$

Figure 6.9 illustrates the efficiency of KERMIT on-line automatic tuning as a function of data scale. In this case, KERMIT efficiency is defined as the percentage of the best possible tuning performance achieved by KERMIT. For small data scales, KERMIT efficiency is about 46% at 300 GB. However, KERMIT efficiency increases significantly with increasing data scale, reaching 92.7% at 2 TB. At larger data scales, KERMIT on-line automatic tuning is expected to be even closer to the best possible tuning.

There can be many reasons to explain why KERMIT efficiency seems to improve with an increasing data scale.

- A larger data scale means a higher number of containers and, with that, there is more information to make the choice on what parameters offer the best performance under the current conditions.
- More tuning waves are also a direct result of increasing data scale; this increases the odds of having a good configuration and correcting a possible drop in performance with a local search.

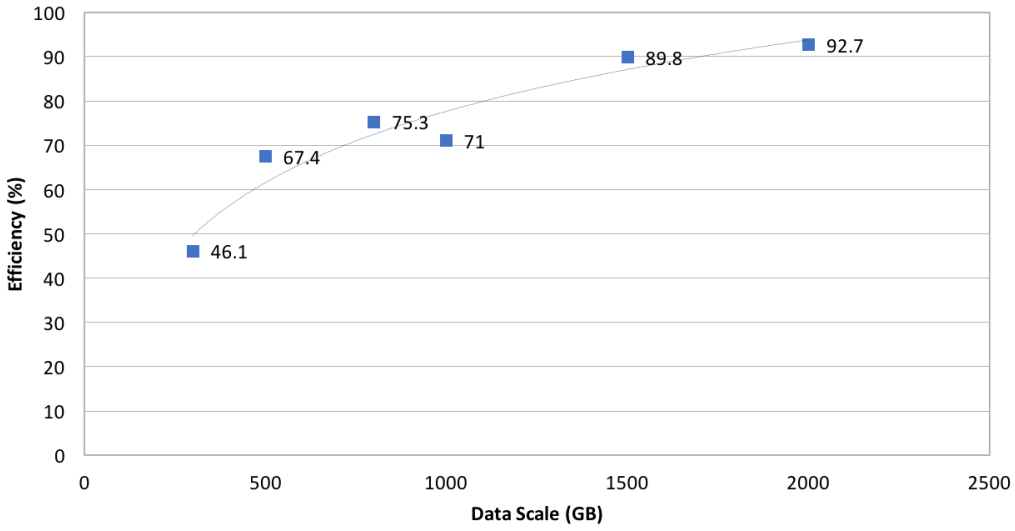

Figure 6.9: KERMIT efficiency (percentage of the best possible tuning performance achieved by KERMIT) at different data scales.

- The relative cost of the early tuning waves that are used to explore the search space is also reduced proportionally when the data scale is increased. This is because the number of tuning waves used to explore the cost options and to find the optimal configuration is the same for a small or a large data scale.

Chapter 7

Conclusions

7.1 Introduction

In this chapter the results achieved through this research work will be reviewed. The decisions regarding the creation and enhancement of KERMIT will be discussed along with the conclusions that can be drawn from the data itself.

7.2 Decision Analysis

In this section, decisions made by the research team will be analyzed along with the reasons for making those decisions, and what can be inferred from the results obtained

7.2.1 Tuning YARN

The team decided to do tuning based on the YARN framework to be able to gain flexibility by making KERMIT tuning available to platforms beyond Hadoop. This was a part of the original design of KERMIT from the beginning and even before the author of this thesis joined the team. This decision has proven to work in practice because it is also possible to use KERMIT automatic tuning in Spark applications as was demonstrated by the results discussed previously regarding Spark tuning.

7.2.1.1 Tuning Spark by tuning YARN

It is worth pointing out that it was always considered to be possible to use KERMIT to tune Spark by tuning YARN; however, this was actually demonstrated to be possible.

The results of tuning YARN are not currently showing gains on the same scale as the results obtained in doing tuning for Hadoop; this is for several reasons:

- The default Spark settings for memory and virtual cores assigned to containers were found to be the best settings in the exploration done in the search space. This resulted in a case where the default settings or the shallow tuning delivered optimal performance; it is very unlikely to see that repeated in a large cluster used for production in an organization.
- The containers in Spark have a very different life cycle than those created in Hadoop; Spark containers are created at the beginning. This means that the team's current method of tuning YARN is not going to be able to do tuning for a single Spark job. It is possible to do tuning if there is a series of Spark jobs being executed one after another or in a simultaneous manner. This may result in more complex behavior that can be harder to tune.

7.2.1.2 Conclusion

In retrospect, we can conclude that the decision to tune YARN was correct because it enabled KERMIT to work in more environments than other automatic tuning efforts like MRONLINE. It is also relevant to recognize that, by broadening the scope of KERMIT tuning, the complexity of the problem can also increase because there can be differences in how YARN works inside different frameworks.

7.2.2 Adding Enhanced Logging Infrastructure

To reach the point where the team could perform large runs and compare them, it was necessary to have the tools to easily extract relevant information. When appropriate tools were not present, the process of knowing what was happening inside KERMIT was a matter of manually looking at the Resource Manager log file. This process was not efficient because that log file normally has a lot of information coming from the regular execution of YARN that is not relevant to understanding the tuning process.

Having a file that contains only the information that the KERMIT component sends to the log makes the process of seeing what is happening inside KERMIT easier. This translated into a better understanding of the internal working of the YARN jobs.

To analyze all the runs discussed using different data sizes and benchmarks would be nearly impossible without the tools that extract the data automatically from the KERMIT log. Being able to use those tools to obtain statistics allowed the team to process and compare data at a much higher pace than before.

These enhancements are the basis of some of the improvements the research team is working on for future work; this will be elaborated upon in the next chapter.

7.2.3 Changing the Tuning Approach

Before the author of this thesis became a part of the research team, KERMIT was using an implementation of Hill Climbing as the main tuning algorithm. The use of Hill Climbing was motivated by the fact that the algorithm was considered very good at finding global optimum values and at avoiding getting stuck in local optimum values. After running experiments using Hill Climbing and analyzing the results, it became clear that the tuning was not obtaining sufficient improvement to reduce the running time. The main problem was that it was taking too many tuning waves to find an optimum value; this resulted in excessive overhead.

After gathering the data from some exploratory memory search space runs, it became clear that the search space was not very complex. This motivated the research team to try to find another tuning algorithm that would need fewer tuning waves, even at the price of sacrificing some of the good aspects of Hill Climbing such as an exhaustive exploration of the search space that would be very likely to avoid any local optima. This led to the creation of the new version of Explorer that, in this research work, has been called Explorer V2. This algorithm would iterate through the search space using some user-predefined values that must be spread out through the search space. This was previously illustrated in the Figure 5.1. This, coupled with the fact that Explorer V2 carried out just one global search, and thereafter only local searches were possible, limited costly exploration even further.

This new version of Explorer was used as the tuning algorithm for the experiments mentioned in this research work. The performance obtained was better than in the previous tests using Hill Climbing because there was an improvement in terms of total time using Explorer V2. The improvement was proportionally larger when the jobs were on an enhanced scale as was discussed in the previous chapter; this encourages us to think KERMIT can be useful for tuning in the real-world execution of very big data jobs.

7.2.3.1 The Future of Hill Climbing in KERMIT

In this research work, we replaced Hill Climbing with a simpler, more straightforward algorithm that served us better for the task at hand. We want to consider types of Hill Climbing for the future improvements that we want to make to KERMIT, especially the more efficient Smart Hill Climbing. In the future we might add more variables to tune and Explorer V2 will then not likely be the most efficient way to explore. An improved version of Hill Climbing can also be useful to make the tuning process more resilient to statistical noise that can deceive the algorithm and result in bad configuration values being used.

7.3 Results Analysis

It is considered that the data scale that was used in this study of up to 2 TB was a good indicator that KERMIT can perform efficiently in real-world big data applications. This was one of the team's goals because MRONLINE [25], the other research project that dealt with automatic tuning of Hadoop/YARN, used data sizes of a maximum of 100 GB; this would be considered small if compared to leading benchmarks like TPCx-HS [54] that have 1 TB as the smallest data scale allowed for that specific benchmark. The data scale is of particular interest; the results at different data scales are not comparable to each other because the computational challenges can vary considerably at different data volumes.

7.4 Potential Uses for KERMIT

The author believes that automatic online tuning is a better option than manual or offline tuning in many cases. It is clear that some configuration parameters would not be feasible options to include in an online tuning approach; on the other hand, automatic tuning can be used to tune parameters that are common among applications like memory, virtual cores, network utilization and disk utilization allotments.

It is possible to imagine a situation where KERMIT and other auto tuning tools such as MRONLINE can work together in a YARN cluster. In that case, KERMIT would tune the common parameters, MRONLINE would tune the application specific parameters, and the user would be responsible of the remaining configuration parameters. This is one potential use for KERMIT; however, KERMIT can also improve

the performance of shallow tuning by itself as was shown in the experiments.

Chapter 8

Future Work

8.1 Introduction

In this chapter the research that is planned for the future by the team will be discussed. Some of these topics are in a very early stage of planning, and others have been partially implemented. All of these topics aim to improve or expand KERMIT and further the studies the team has been conducting on YARN tuning and performance.

8.2 Running KERMIT on a New Cluster

A new larger cluster is being configured, to be used to execute KERMIT; this cluster will have more nodes and will run on higher-end hardware. This cluster also has a different storage architecture that will be an interesting subject for further studies.

The cluster specification is: an eight-node cluster comprising one management node and seven data nodes (all virtual machines running on Power PC 64 Little Endian (PPC64 LE)). Each node is equipped with one virtual disk with 100 GB capacity running on a solid-state drive (SSD) for the operating system and Hadoop/Spark stack installation. Each node has access to a 12 TB shared drive based on SSD storage; this is used as the HDFS storage and temporary directories. Each node is also equipped with 49 GB of RAM and 10 PPC64 LE CPUs running at 3.4 GHz, with one core per CPU. All nodes are running the Ubuntu 16.04 Long Time Support (LTS) operating system.

This architecture is especially interesting because using a shared drive for all the nodes based on SSD is something that has not yet been done in a research study.

This much more capable setup in terms of processing power will provide the team with the capacity for running bigger and more complex runs. The CPU computing power will also make it possible to explore in a much more profound way the impact that can be achieved by tuning the virtual cores assigned to the YARN containers.

8.3 Virtual Cores Tuning

The team decided to focus its efforts on exploring the possibilities of improving YARN performance by doing automatic tuning to the memory assigned to YARN containers. Time constraints forced us to choose which kind of parameter would be the primary focus. It was important to make the decision based on which parameter would be most likely to obtain good performance improvement. It was assumed that the physical CPUs present in the cluster were not powerful enough to make the virtual cores value a major factor in cluster performance.

When the new cluster setup is ready to be used, the team will make runs to assess the impact of tuning the number of virtual cores; the additional computing power provided by the processors used in that cluster could make the number of virtual cores a parameter with a big impact on cluster performance.

8.4 Docker Implementation

Exploring the use of cloud containers such as Docker to run YARN applications is an interesting alternative to apply our tuning efforts. Docker containers are an attractive alternative to using JVMs to host YARN containers because they can provide shorter provisioning time, better performance isolation, higher resource utilization and bypass the virtual IO [56]. There is a study that claims that Docker containers can provide superior performance compared to JVMs [57]. Docker containers could help the team to improve the performance gains achieved by KERMIT, or they could reduce the statistical noise seen in the container statistics obtained inside KERMIT. Experiments using Docker, or a similar cloud container technology, will be likely carried out in the future.

8.5 Workload Characterization

The KERMIT Autotuner currently works by analyzing the YARN containers duration and it can be considered totally agnostic to internal phases and tasks inside a YARN job. It would be possible to do a more in depth analysis of the container statistics and how they behave during the different internal tasks of some of the most important jobs. The points where a YARN job transitions from one internal task to another could be very good moments to prepare to search for better parameters because the new internal task may have different requirements in terms of storage and processing capabilities.

There is a plan to gather more comprehensive statistics and to analyze them to try to understand better how tuning could be improved. This could help the KERMIT Autotuner make better decisions regarding whether it is appropriate or not to search for better parameter values.

List of References

[1] A. Arora and S. Mehrotra, *Learning YARN: moving beyond MapReduce–learn resource management and big data processing using YARN*. Birmingham, UK: Packt Publishing, 2015.

[2] M. Pospelova, C. U. Theses, and D. C. Science, *Real Time Autotuning for MapReduce on Hadoop/YARN*. Ottawa: Carleton University, 2015.

[3] H. Herodotou, H. Lim, G. Luo, N. Borisov, L. Dong, F. B. Cetin, and S. Babu, "Starfish: A self-tuning system for big data analytics.," in *CIDR*, vol. 11, pp. 261–272, 2011.

[4] J. Shi, J. Zou, J. Lu, Z. Cao, S. Li, and C. Wang, "MRTuner: a toolkit to enable holistic optimization for mapreduce jobs," *Proceedings of the VLDB Endowment*, vol. 7, no. 13, pp. 1319–1330, 2014.

[5] M. Genkin, F. Dehne, M. Pospelova, Y. Chen, and P. Navarro, "Automatic, on-line tuning of yarn container memory and cpu parameters," in *High Performance Computing and Communications; IEEE 14th International Conference on Smart City; IEEE 2nd International Conference on Data Science and Systems (HPCC/SmartCity/DSS), 2016 IEEE 18th International Conference on*, pp. 317–324, IEEE, 2016.

[6] G. Turkington, S. Karanth, and T. Deshpande, *Hadoop: data processing and modelling : unlock the power of your data with Hadoop 2.X ecosystem and its data warehousing techniques across large data sets*. Birmingham, UK: Packt Publishing, 2016.

[7] H. Karau, A. Konwinski, and P. Wendell, *Learning Spark*. O'Reilly Media, 1 ed., 2015.

[8] S. Ghemawat, H. Gobioff, and S.-T. Leung, "The google file system," in *ACM SIGOPS operating systems review*, vol. 37, pp. 29–43, ACM, 2003.

[9] J. Dean and S. Ghemawat, "Mapreduce: simplified data processing on large clusters," *Communications of the ACM*, vol. 51, no. 1, pp. 107–113, 2008.

[10] K. K.C., *Performance Tuning of MapReduce Programs*. PhD thesis, North Carolina State University, 2015.

[11] M. Li, *A resource management framework for cloud computing*. PhD thesis, Virginia Polytechnic Institute and State University, Virginia Polytechnic Institute and State University, 2014.

[12] M. Khan, *Hadoop performance modeling and job optimization for big data analytics*. PhD thesis, Brunel University London, 2015.

[13] T. Lipcon, "Cloudera: 7 tips for improving mapreduce performance," 2009.

[14] T. Lipcon, "Cloudera: Optimizing mapreduce job performance," 2012.

[15] Impetus, "Hadoop performance tuning," 2012.

[16] C. A. Schaefer, V. Pankratius, and W. F. Tichy, "Atune-il: An instrumentation language for auto-tuning parallel applications," in *Euro-Par 2009 Parallel Processing*, pp. 9–20, Springer, 2009.

[17] M. J. Cafarella and C. Ré, "Manimal: relational optimization for data-intensive programs," in *Proceedings of the 13th International Workshop on the Web and Databases*, p. 10, ACM, 2010.

[18] J. Liu, N. Ravi, S. Chakradhar, and M. Kandemir, "Panacea: towards holistic optimization of mapreduce applications," in *Proceedings of the Tenth International Symposium on Code Generation and Optimization*, pp. 33–43, ACM, 2012.

[19] Q. Chen, M. Guo, Q. Deng, L. Zheng, S. Guo, and Y. Shen, "Hat: history-based auto-tuning mapreduce in heterogeneous environments," *The Journal of Supercomputing*, vol. 64, no. 3, pp. 1038–1054, 2013.

[20] H. Herodotou and S. Babu, "Profiling, what-if analysis, and cost-based optimization of mapreduce programs," *Proc. of the VLDB Endowment*, vol. 4, no. 11, pp. 1111–1122, 2011.

[21] G. Liao, K. Datta, and T. L. Willke, "Gunther: search-based auto-tuning of mapreduce," in *Euro-Par 2013 Parallel Processing*, pp. 406–419, Springer, 2013.

[22] B. Behzad, S. Byna, S. M. Wild, M. Prabhat, and M. Snir, "Improving parallel i/o autotuning with performance modeling," in *Proceedings of the 23rd International Symposium on High-performance Parallel and Distributed Computing*, HPDC '14, (New York, NY, USA), pp. 253–256, ACM, 2014.

[23] N. Yigitbasi, T. L. Willke, G. Liao, and D. Epema, "Towards machine learning-based auto-tuning of mapreduce," in *Modeling, Analysis & Simulation of Computer and Telecommunication Systems (MASCOTS), 2013 IEEE 21st International Symposium on*, pp. 11–20, IEEE, 2013.

[24] E. R. Lucas Filho, E. C. De Almeida, Y. Le Traon, *et al.*, "Intra-query adaptivity for mapreduce query processing systems," in *IDEAS 2014: 18th International Database Engineering & Applications Symposium*, 2014.

[25] M. Li, L. Zeng, S. Meng, J. Tan, L. Zhang, A. R. Butt, and N. Fuller, "MRONLINE: MapReduce Online Performance Tuning," in *Proceedings of the 23rd International Symposium on High-performance Parallel and Distributed Computing*, (New York, NY, USA), pp. 165–176, ACM, 2014.

[26] K. Kc and V. W. Freeh, "Dynamically Controlling Node-Level Parallelism in Hadoop," in *Cloud Computing (CLOUD), 2015 IEEE 8th International Conference on*, pp. 309–316, IEEE, 2015.

[27] B. Xi, Z. Liu, M. Raghavachari, C. H. Xia, and L. Zhang, "A smart hill-climbing algorithm for application server configuration," in *Proceedings of the 13th international conference on World Wide Web*, pp. 287–296, ACM, 2004.

[28] R. Thonangi, V. Thummala, and S. Babu, "Finding good configurations in high-dimensional spaces: Doing more with less," in *Modeling, Analysis and Simulation of Computers and Telecommunication Systems, 2008. MASCOTS 2008. IEEE International Symposium on*, pp. 1–10, IEEE, 2008.

[29] Tao Ye, H. Kaur, S. Kalyanaraman, and M. Yuksel, "Large-Scale Network Parameter Configuration Using an On-Line Simulation Framework," *IEEE/ACM Transactions on Networking*, vol. 16, no. 4, pp. 777–790, 2008.

[30] X.-S. Yang, "Nature-Inspired Algorithms: Success and Challenges," in *Engineering and Applied Sciences Optimization*, vol. 38, pp. 129–143, Springer International Publishing, 2015.

[31] C. A. Tovey, "Hill climbing with multiple local optima," *SIAM Journal on Algebraic Discrete Methods*, vol. 6, no. 3, pp. 384–393, 1985.

[32] S. Herrmann and F. Rothlauf, "Predicting heuristic search performance with pagerank centrality in local optima networks," in *Proceedings of the 2015 on Genetic and Evolutionary Computation Conference*, pp. 401–408, ACM, 2015.

[33] C. Carpineto and G. Romano, "Optimal meta search results clustering," in *Proceedings of the 33rd international ACM SIGIR conference on Research and development in information retrieval*, pp. 170–177, ACM, 2010.

[34] K. Patrick, "Comparison of simulated annealing and hill climbing in the course timetabling problem," *African Journal of Mathematics and Computer Science Research*, vol. 5, no. 11, pp. 176–178, 2012.

[35] T. Ye and S. Kalyanaraman, "A recursive random search algorithm for network parameter optimization," *ACM SIGMETRICS Performance Evaluation Review*, vol. 32, no. 3, pp. 44–53, 2004.

[36] C. J. Price, M. Reale, and B. L. Robertson, "A cover partitioning method for bound constrained global optimization," *Optimization Methods and Software*, vol. 27, no. 6, pp. 1059–1072, 2012.

[37] K. Wang, X. Lin, and W. Tang, "Predator—An experience guided configuration optimizer for Hadoop MapReduce," in *Cloud Computing Technology and Science (CloudCom), 2012 IEEE 4th International Conference on*, pp. 419–426, IEEE, 2012.

[38] S. Kirkpatrick, "Optimization by simulated annealing: Quantitative studies," *Journal of statistical physics*, vol. 34, no. 5-6, pp. 975–986, 1984.

[39] N. Metropolis, A. W. Rosenbluth, M. N. Rosenbluth, A. H. Teller, and E. Teller, "Equation of State Calculations by Fast Computing Machines," *The Journal of Chemical Physics*, vol. 21, no. 6, p. 1087, 1953.

[40] D. K. Hale, B. B. Park, A. Stevanovic, P. Su, and J. Ma, "Optimality versus run time for isolated signalized intersections," *Transportation Research Part C: Emerging Technologies*, vol. 55, pp. 191–202, 2015.

[41] A. Misevičius, T. Blažauskas, J. Blonskis, and J. Smolinskas, "An overview of some heuristic algorithms for combinatorial optimization problems," *Information Technology and Control*, vol. 30, no. 1, 2004.

[42] E. Roux, A. Ramalli, P. Tortoli, C. Cachard, M. Robini, and H. Liebgott, "Speed-up of acoustic simulation techniques for 2d sparse array optimization by simulated annealing," in *Ultrasonics Symposium (IUS), 2015 IEEE International*, pp. 1–4, IEEE, 2015.

[43] J. H. Holland, *Adaptation in natural and artificial systems: an introductory analysis with applications to biology, control, and artificial intelligence*. MIT press, 1992.

[44] A. Rozsa, A. Glock, and T. Boult, "Genetic Algorithm Attack on Minutiae-Based Fingerprint Authentication and Protected Template Fingerprint Systems," in *Proceedings of the IEEE Conference on Computer Vision and Pattern Recognition Workshops*, pp. 100–108, 2015.

[45] F. Glover, "Future paths for integer programming and links to artificial intelligence," *Computers & operations research*, vol. 13, no. 5, pp. 533–549, 1986.

[46] A. M. Connor, K. A. Seffen, G. T. Parks, and P. J. Clarkson, "Efficient optimisation of structures using tabu search," *Proceedings of the 1st ASMO/ISSMO Conference on Engineering Design Optimization*, 2014.

[47] G. Nápoles, I. Grau, and R. Bello, "Particle Swarm Optimization with Random Sampling in Variable Neighbourhoods for Solving Global Minimization Problems," in *Swarm Intelligence*, vol. 7461, pp. 352–353, Berlin, Heidelberg: Springer Berlin Heidelberg, 2012.

[48] B. L. Miller and D. E. Goldberg, "Optimal sampling for genetic algorithms," *Urbana*, vol. 51, p. 61801, 1996.

[49] B. Saha, H. Shah, S. Seth, G. Vijayaraghavan, A. Murthy, and C. Curino, "Apache tez: A unifying framework for modeling and building data processing applications," in *Proceedings of the 2015 ACM SIGMOD International Conference on Management of Data*, SIGMOD '15, (New York, NY, USA), pp. 1357–1369, ACM, 2015.

[50] R. Zhang, M. Li, and D. Hildebrand, "Finding the big data sweet spot: Towards automatically recommending configurations for hadoop clusters on docker containers," in *2015 IEEE International Conference on Cloud Engineering*, pp. 365–368, March 2015.

[51] T. Ivanov and M.-G. Beer, "Performance evaluation of spark sql using bigbench," in *Workshop on Big Data Benchmarks*, pp. 96–116, Springer, 2015.

[52] D. Eadline, *Hadoop 2 Quick-Start Guide: Learn the Essentials of Big Data Computing in the Apache Hadoop 2 Ecosystem*. Addison-Wesley Professional, 1st ed., 2015.

[53] A. S. Foundation, "Terasort description," 2017.

[54] R. Nambiar, M. Poess, A. Dey, P. Cao, T. Magdon-Ismail, A. Bond, *et al.*, "Introducing tpcx-hs: the first industry standard for benchmarking big data systems," in *Technology Conference on Performance Evaluation and Benchmarking*, pp. 1–12, Springer, 2014.

[55] M. Genkin, "Goals and challenges of the spark multiuser benchmark," 2016.

[56] R. Zhang, M. Li, and D. Hildebrand, "Finding the big data sweet spot: Towards automatically recommending configurations for hadoop clusters on docker containers," in *Cloud Engineering (IC2E), 2015 IEEE International Conference on*, pp. 365–368, IEEE, 2015.

[57] W. Felter, A. Ferreira, R. Rajamony, and J. Rubio, "An updated performance comparison of virtual machines and linux containers," in *Performance Analysis of Systems and Software (ISPASS), 2015 IEEE International Symposium On*, pp. 171–172, IEEE, 2015.