2016 IEEE 18th International Conference on High Performance Computing and Communications; IEEE 14th International Conference on Smart City; IEEE 2nd International Conference on Data Science and Systems

Automatic, on-line tuning of YARN container memory and CPU parameters

Mikhail Genkin*, Frank Dehne, Maria Pospelova, Yabing Chen and Pablo Navarro
School of Computer Science, Carleton Unviersity, Ottawa, Canada
* Contact Author. E-mail: michael.genkin@carleton.ca

***Abstract*—Big data analytic technologies such as Hadoop and Spark run on compute clusters that are managed by resource managers such as YARN. YARN manages resources available to individual applications, thereby affecting job performance. Manual tuning of YARN tuning parameters can result in sub-optimal and brittle performance. Parameters that are optimal for one job may not be well suited to another. In this paper we present KERMIT, the first on-line automatic tuning system for YARN. KERMIT optimizes in real-time YARN memory and CPU allocations to individual YARN containers by analyzing container response-time performance. Unlike previous automatic tuning methods for specific systems such as Spark or Hadoop, this is the first study that focuses on the more general case of on-line, real-time tuning of YARN container density and how this affects performance of applications running on YARN. KERMIT employs the same tuning code to automatically tune any system that uses YARN, including both Spark and Hadoop. The effectiveness of our technique was evaluated for Hadoop and Spark jobs using the Terasort, TPCx-HS, and SMB benchmarks. KERMIT was able to achieve an efficiency of more than 92% of the best possible tuning configuration (exhaustive search of the parameter space) and up to 30% faster than basic manual tuning.**

Keywords: Automatic tuning, on-line tuning, YARN, Hadoop, Spark.

I. Introduction

Big data analytics has emerged as one of the most important computing trends in high performance computing. Key big data technologies such as Hadoop map-reduce jobs, Spark applications, Hive, Hbase and others run on clusters that are managed by resource managers. The most commonly used resource manager in the big data analytics space is YARN. It manages resources available to individual applications, thereby affecting job performance. Manual tuning of big data applications, such as Hadoop map-reduce or Spark, involves changing many YARN tuning parameters. It often results in sub-optimal and brittle performance because parameters that are optimal for one job (input data set) may not be well suited to another.

There have been a number of attempts to automatically tune big data applications [1], [2], [3], [4], but they focused specifically on tuning Hadoop map-reduce only. This is the first study that focuses on the more general case of on-line, real-time tuning of YARN container density and how this affects performance of applications running on YARN. The parameter ”container density” refers to the number of containers started by YARN at a given point in time to run in parallel on the cluster. Running too few containers may leave the cluster under-utilized, resulting in sub-optimal job performance. Running too many containers at the same time can create contention for key resources such as CPU and memory, again leading to poor application performance.

As part of our study we developed an on-line automatic tuning Java library, called KERMIT, designed specifically for on-line automatic tuning. Although in this study KERMIT was integrated with YARN, it exposes an integration interface that makes it easy for any application or resource manager to integrate with it.

In this paper we demonstrate that KERMIT can optimize performance of both Hadoop map-reduce jobs and Spark applications using the same integration code and search algorithm. KERMIT optimizes the number of concurrently running containers on-line, in real-time, in response to actual observed container performance, and can achieve a performance of more than 92% of the best possible tuning configuration (exhausive search of parameter space). Our results demonstrate that on-line automatic tuning can achieve tuning configurations that are up to 30% faster than the basic manual tuning typically applied by big data practitioners. Our technique does not require extensive, time-consuming training runs. It also has a very small code footprint and performance overhead. It does not require any application-specific tuning rules to be developed by application administrators. KERMIT allows application frameworks such as Hadoop map-reduce and Spark to dispense with many manual configuration settings, thereby reducing coding effort for application framework developers and improving end-user experience for administrators.

The remainder of this paper is organized as follows. Section II reviews previous work on automatic tuning of big data workloads and discusses key differences between previous work and our approach. Section III first discusses the YARN architecture and how YARN parameters can impact performance of Hadoop map-reduce, Spark and other applications. Section III then describes our KERMIT architecture and how it integrates with YARN. Section IV explains the experimantal evaluation methodology, including criteria used to quantify performance improvements and automatic tuning efficiency. Section V presents our experimental results. Section VI summarizes our key findings and discusses directions for future research.

978-1-5090-4297-5/16 $31.00 © 2016 IEEE
DOI 10.1109/HPCC-SmartCity-DSS.2016.73

II. Previous Work

There have been many attempts to automatically tune performance of big data workloads. Most of them focused on automatically tuning Hadoop map-reduce. None of the previous papers studied the more general case of auto-tuning a general purpose resource manager such as YARN. In this section we discuss relevant related work.

Performance of Hadoop jobs is sensitive to underlying hardware, network infrastructure, JVM configuration and Hadoop parameter settings. These challenges had been addressed by multiple research groups, revealing the importance of each of the levels [5], [3], [2], [6]. While issues related to underlying hardware and network infrastructure could be addressed via intelligent infrastructure design, impacts related to JVM and Hadoop configuration must be attended each time a new map-reduce job is executed. Apache Hadoop exposes over 200 tunable parameters. About 10% of these have significant impact on performance [7], [1], [6]. Manually tuning performance of Hadoop map-reduce jobs involves trying to experimentally establish the ideal mix of settings for each new job. This is a time-consuming and labor-intensive process. A number of companies specializing on Hadoop deployments have proposed Rule of Thumb (RoT) tuning guides [8], [9], [10]. The difference in job duration (response-time) between commonly used RoT settings and optimal configuration can be up to 100 fold [4], [6].

Automatic tuning of Hadoop was explored by multiple research groups. Schaefer and colleagues [5] developed an automatic tuning language for Map Reduce, Atune-IL, that allows a user to explore a selected set of tunable variables, such as the number of threads or numbers of map and reduce jobs, within defined user limits. Atune-IL also allows a user to explore the possible performance impact of alternative code block implementations. This approach automates manual tuning to some extent, reduces search space, and translates the solution search space to a simpler tool. However, it requires a user to have Hadoop tuning expertise in order to select important tunables, and to set up limits and the size of the step for each tunable. The main drawback to Atune-IL is that it requires a developer to learn a new syntax, specific to the language.

HAT, a history-based automatic tuning framework for map-reduce, was introduced by Chen and colleagues in 2013 [11]. HAT tunes the weight of each stage of a map and a reduce task according to values of the tasks in the history. It orchestrates Hadoop back up task execution according to current and historical weights of the tasks. The authors claimed 37% job execution time improvement.

Starfish is a cost-based Hadoop automatic tuning framework developed at Duke University by Babu and colleagues [1], [2]. The authors succeeded in applying a cost-based approach, popular in relational databases, to optimizing map-reduce performance. Starfish takes into account different stages of a map-reduce 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 map-reduce tunables.

In 2013, Liao and colleagues analyzed model-based approaches for Hadoop map-reduce optimization and identified a number of major limitations [6]. The research group proposed and implemented Gunther - a search-based automatic tuner for Hadoop map-reduce. The authors evaluated several global search algorithms, and selected a genetic algorithm (GA) for their search implementation. Their modified GA was evaluated on two clusters. Experimental results demonstrated that Gunther achieved near-optimal performance in a small number of trials (<30), and yielded better performance improvements than rule-of-thumb tuning and cost-based automatic tuning approaches.

A machine learning approach was explored by Yigitbashi and colleagues [7], who analyzed various machine learning-based performance models. Their analysis was conducted on two common Hadoop applications - Terasort and word count - with data sets of various sizes. These authors came to the conclusion that support vector regression exhibits the best performance among machine learning methods. The support vector regression-based automatic tuner was shown to outperform Starfish.

Another search-based evolutionary computation approach was studied by Filho and colleagues in 2014 [12]. They proposed an adaptive tuning mechanism that enabled the setting 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.

All previous work discussed so far are examples of off-line tuning approaches, where changes to configuration settings are made before or after job execution. An on-line automatic tuning approach was explored by Li et al. in MRONLINE [13]. MRONLINE automatically tunes map-reduce job performance as the job runs. This approach has a number of advantages. Each map-reduce job, depending on the type of computation it performs, can respond differently to changes in tuning parameters. Therefore relying on historical information accumulated from previous jobs as in [11] can yield sub-optimal results. Approaches that rely on modeling [1], [2], or large numbers of training runs [6], are labor-intensive and expensive. MRONLINE focuses on automatically tuning several important parameters of a map-reduce job as it runs. In this respect MRONLINE is the most closely related previous work to our study. MRONLINE focuses on changing map-reduce configuration settings for individual job tasks and correlating these changes with individual task completion times. It worked at the Hadoop map-reduce framework level. MRONLINE introduced a specialized search algorithm, called gray-box hill climbing. The algorithm was designed to reduce the number of search iterations needed to find a good-enough solution by allowing the end-user to configure a set of rules that effectively restricted the search space to speed up solution convergence.

MRONLINE included a monitor, an on-line tuner, and a dynamic configurator. Rather than reading configuration parameters from the mapred-site.xml file, map and reduce tasks received their CPU and memory parameters from the dynamic configurator. Different tasks could receive different CPU and memory settings, and could thus display different completion times. The monitor would keep track of task execution times and feed this information to the on-line tuners. The on-line tuner would use a gray-box hill climbing algorithm to find the next set of parameters for subsequent tasks. MRONLINE performance was evaluated using Terasort and BBP benchmarks at data scales up to 100GB. The authors claimed 30% performance improvement relative to the default configuration[13].

III. KERMIT Automatic Tuning Architecture

Our KERMIT system takes an on-line approach similar to that used in MRONLINE but with important differences. MRONLINE introduced dynamic configuration into the Hadoop MapReduce framework. KERMIT introduces dynamic configuration into the more general YARN resource manager framework. Unlike MRONLINE, KERMIT does not keep track of task performance, but focuses instead on YARN container life-cycle. KERMIT runs as an integral part of the YARN resource manager, intercepts resource requests arriving from frameworks, and adjusts actual memory and CPU settings used for containers. This allows KERMIT to be used not only with Hadoop MapReduce, but also with other analytic frameworks such as Spark. In KERMIT we also implement a different search algorithm that does not require users to configure domain-specific rules. KERMIT is capable of finding close-to-optimal solutions in a small number of iterations. Compared to MRONLINE, KERMIT was evaluated using a more complex benchmarks, such as TPCx-HS, using much larger and more realistic data scales up to 2 TB. KERMIT also introduces a new, low-overhead on-line tuning algorithm called the Explorer.

We first review the relevant aspects of the YARN architecture, and then present the details of our KERMIT method.

A. YARN Architecture Review

For ease of presentation we discuss YARN within the context of Hadoop MapReduce v.2.0 which uses YARN but our discussion applies to other YARN frameworks as well. The YARN ResourceManager has two main components - the Scheduler and the ApplicationsMaster. The Scheduler is responsible for allocating resource to various applications running on the cluster. The scheduler uses an abstract Container to encapsulate all tasks and applications. The Container incorporates key system resources needed by the applications - in Hadoop 2.6.0 it supports memory and CPU. The YARN architecture is shown in Figure 1.

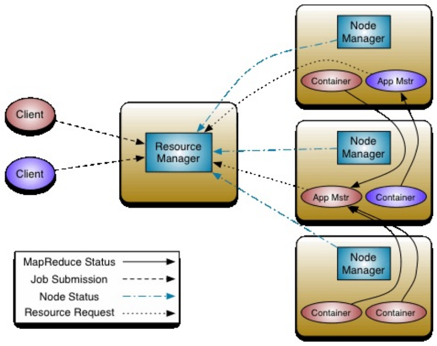

Fig. 1. YARN architecture [14].

The ApplicationsManager component of the ResourceManager is responsible for accepting job-submissions, negotiating the first container for executing the application specific ApplicationMaster. The NodeManager is responsible for containers, monitoring their resource usage (cpu, memory in Hadoop 2.6.0) and reporting it to the ResourceManager/Scheduler.

The per-application ApplicationMaster has the responsibility of negotiating appropriate resource containers from the Scheduler, tracking their status and monitoring for progress.

B. KERMIT Architecture

Figure 2 shows the KERMIT architecture and how it relates to YARN. YARN is designed to support multiple frameworks. Frameworks, such as Apache Hadoop MapReduce and Apache Spark connect to the YARN resource manager server. When a framework, for example Hadoop MapReduce, needs to run a job it sends a resource request to the YARN resource manager. The resource request includes all of the remaining tasks that need to be run and the desired amount of memory and CPU for each task. For Hadoop MapReduce the desired amount of memory and CPU for each task in the request is determined by the values in mapred-site.xml for: (1) mapreduce.map.memory.mb; (2) mapreduce.map.cpu.vcores; (3) mapreduce.reduce.memory.mb; (4) mapreduce.reduce.cpu.vcores. The YARN resource manager then uses one of a number of configurable schedulers to determine how much memory and CPU is available on the cluster data nodes to be allocated to this job and determines how many tasks can be executed concurrently on the cluster. YARN creates a Container to track the execution of each task. The Container in the version of YARN used in this study is a virtual construct used to compartmentalize resource allocations for tasks and track their life-cycle. The number of Containers created by YARN is determined by the ratio of memory and CPU resources requested by the framework to the total number of memory and CPU resources available for this job on the cluster. Thus the values of map-reduce parameters listed above determine the container density. Container density, in turn, governs the number of JVM processes concurrently executing on the cluster. Optimizing the number of concurrently executing JVM processes produces performance improvements.

KERMIT is a Java class library that is loaded by and runs in-process to the YARN resource manager server. Figure 3 shows how KERMIT integrates with the YARN resource manager. KERMIT consists of two main components: (1) AutoTuner;

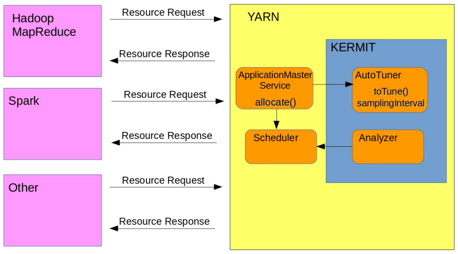

Fig. 2. KERMIT architecture.

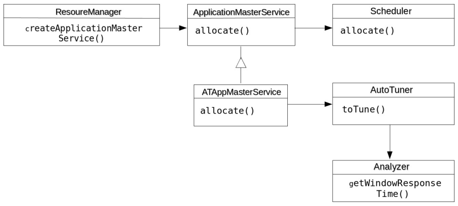

Fig. 3. How KERMIT integrates with YARN.

(2) Analyzer. KERMIT extends the ApplicationMasterService class. It overrides the allocate() method, and intercepts the resource request arriving from the framework. The KERMIT AutoTuner then changes the amount of memory and CPU resources requested by the framework for each task and observes the effect using the Analyzer. This integration technique has a number of advantages. It allows KERMIT to work with any of the available YARN schedulers and application masters. This is possible because in all cases the resource request will go through the allocate() method of the ApplicationMasterService class. The request is application-agnostic and contains only generic desired CPU and memory information. KERMIT focuses on container response-time analysis and is not aware of any application-specific metrics.

The KERMIT AutoTuner has a sampling interval setting called a Window. The Analyzer class maintains a list of Window objects. Each Window object stores container data collected during the corresponding Window sampling interval. These data include the container start time, completion time, and duration. Whenever a container request is made, the Analyzer adds this container's start time to the current Window object. Whenever a container completes, which may happen during one of the subsequent Windows, the Analyzer finds and updates that container's completion time and duration. The Analyzer calculates the average and standard deviation for these container response-time values. Only the last 20% of containers in each Window (i.e. the containers that finished last in that Window) are used for these calculations. This approach is designed to reduce the effects of runtime change adjustments lagging behind dynamic configuration changes.

The AutoTuner uses the Analyzer to detect whether a significant change in container performance has occurred. Significant change is currently defined to occur when the average container duration in the current Window is further apart from the average container duration in the previous Window than the combined container duration standard deviations in those Windows. Once the AutoTuner decides that this significant change has occurred, it uses the Explorer algorithm to perform a search.

The key goals for our Explorer algorithm design were: (1) minimize tuning overhead by restricting the search space; (2) minimize administration and configuration. Explorer uses preconfigured ranges for it's container memory and CPU values - thus restricting the search space significantly, and reducing the number of search iterations. The administrator needs to only enter values for container memory and CPU ranges and specify the duration of the Window. Values for memory and CPU ranges can be generic values picked based on total memory and CPUs available on data node servers. They can be re-used for different applications.

The Explorer algorithm operates using 3 states: (1) Global search; (2) Local search; (3) Observe. It uses container statistics from the Analyzer to trigger state transitions. State transitions include: (1) Initial to Global search; (2) Global search to Observe; (3) Observe to Local search; (4) Local search to Observe. The algorithm spends most of the time in the Observe state, transitioning to Local search state only when significant change in container performance is detected as discussed above.

Note that for on-line tuning, it is important to minimize tuning overhead. Therefore, a simple optimization method with a small number of steps is more desirable. To reduce tuning overhead, the KERMIT AutoTuner performs the Global search only initially (whenever the resource manger is restarted). Local search is performed only when it detects a change in container performance. The rest of the time it simply observes.

Explorer starts with the default memory and CPU settings for Hadoop and Spark and performs a Global search. During Global search, the Explorer adjusts either memory or CPU value one step up or down in the range. Which dimension to adjust first is chosen randomly. Whether to go up or down is chosen randomly as well. While in Global search state the Explorer cycles through all CPU and memory values defined in the ranges using one set of values for one Window. Once all values have been tried, the Explorer picks the best set of settings from it's statistics table and switches from Global search to Observe state. Each subsequent Window will apply optimal container memory and CPU values determined during Global search.

State transition from the Observe state to Local search state is triggered if the average container duration of the current Window is further apart from the average of the previous Window than combined standard deviations of the current and previous Windows. The Explorer randomly picks a dimension and randomly choses whether to move once step up or down in the range for that dimension. If the change resulted in decreased or equal container performance then Explorer moves in the opposite direction in the same dimension. If the change resulted in improved performance the Explorer moves one more step in the same direction. Explorer will keep moving one step at a time in that direction as long as it sees improvements in container performance. If Explorer detects no change after moving in both directions, then it tries to adjust a different dimension using the same logic. If this does not produce an improvement then Explorer stops tuning and transitions back to the Observe state using the best CPU and memory values found during the Local search state.

IV. Evaluation Methodology

Our evaluation methodology focuses on comparing job end-to-end response-time achieved with KERMIT automatic tuning with two baselines: (1) basic tuning baseline; (2) best possible tuning (exhaustive search of parameter space). The reason for using the first 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 gate the performance of both the baseline and the compare, and to make sure that both the baseline and compare had the same JVM heap size and only automatically tuned YARN parameters differed. This tuning approach is also similar to the type of tuning commonly performed by field practitioners.

To achieve the best possible tuning baseline, Hadoop map-reduce or Spark settings that were being intercepted by KERMIT and automatically tuned on the YARN side, were searched exhaustively. The settings that were tuned on the map-reduce 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. 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 make sure 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 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.

Comparison runs for KERMIT on-line automatic tuning were performed at the same data scale. Several automatic tuning runs were performed to ensure consistency of the results. Comparison focused on three aspects: (1) establishing the improvement compared to the basic tuning baseline; (2) establishing improvement compared to the best possible tuning baseline; (3) establishing how the first two results change with increasing volume of data.

For Hadoop, the Terasort and TPCx-HS benchmarks were chosen as the primary workloads for our comparisons. Terasort is a very commonly used Hadoop benchmarking application. It was chosen due to it's widespread use in the Hadoop community. The TPCx-HS benchmark is a recent industry standard. It is 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 sequence of jobs is executed back-to-back without interruptions. It explicitly disallows manual tuning between the benchmark stages, but it explicitly allows for automatic tuning. Since HS-Gen, HS-Sort and HS-Validate perform very different types of processing, it is difficult 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.

For Spark the recently open-sourced Spark Multi-user Benchmark (SMB) was chosen as the workload. YARN works differently with Spark than with Hadoop. When YARN provides containers to Spark, Spark uses them to start it's executors and schedules all job tasks to run on those executors. Spark executors persist unchanged until the job is finished. Thus for a useful on-line automatic tuning test, we need a multi-job workload. SMB executes concurrently multiple identical Spark Terasort jobs, and reports throughput and job duration statistics for these jobs. For this study we used 2 GB data scale for the Spark Terasort implementation and 10 concurrent users for all tests. Our analysis focused on the throughput metric - the most important metric to analytics users. The number of Spark executors was left at default, automatic setting. Spark 1.6.1 was used with YARN 2.6.0.

All measurements were performed on a 4-node cluster comprising 1 management node and 3 data nodes (all bare metal). Each data node was equipped with 1 SSD for operating system and Hadoop stack installation and 4 1.8 TB data disks. Each data node was also equipped with 32 GB RAM and 1 Intel i7 CPU with 4 physical cores running at 1600 MHz. Half of the virtual CPUs visible at the operating system level were disabled via operating system command to reduce noise caused by I/O wait. All nodes were running the Ubuntu 14.04 operating system.

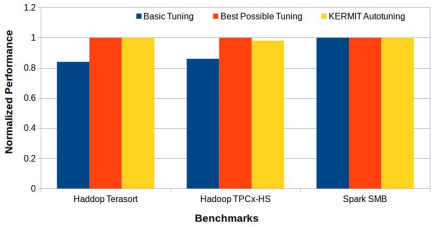

Fig. 4. Normalized performance summary for Hadoop and Spark benchmarks.

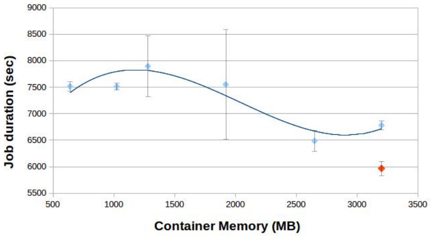

Fig. 5. Data for best possible tuning for Terasort at 500 GB data scale.

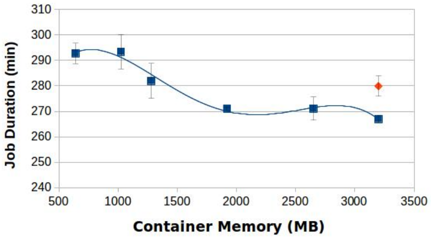

Fig. 6. Data for best possible tuning for TPCx-HS at 500 GB data scale.

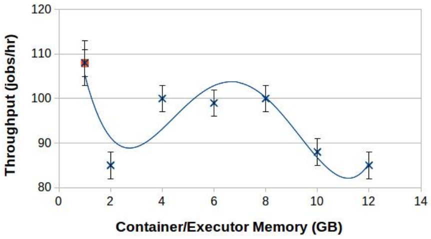

Fig. 7. Data for best possible tuning for SMB.

V. Results

Figure 4 shows a summary of normalized performance achieved by KERMIT for Hadoop and Spark workloads. The best possible tuning result for each benchmark was set equal to 1. For all benchmarks the KERMIT automatic tuning result was observed to be very close to the best possible tuning result. For Hadoop benchmarks Terasort and TPCx-HS, the best possible tuning result was significantly better than the basic tuning result. For the Spark benchmark SMB, our KERMIT Automatic tuning result achieved an optimal level of performance. By 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. In follow-on studies, where we intend to apply this technique to larger clusters and data scales, we expect that this pattern will change and will be similar to what we observe for other workloads such as TPCx-HS.

Figure 5, 6 and 7 show the shape of the search space for Terasort, TPCx-HS, and Spark SMB respectively explored via manual tuning experiments conducted to establish the best possible tuning baseline for container memory. In all cases the data points shown are averages of data points collected at those memory and CPU 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 red 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). For Spark SMB the y-axis records show throughput as measured in jobs per hour. In all cases the search space has a global maximum (within constraints discussed in the Evaluation Methodology section) as well as local maxima and minima. In all cases our Explorer algorithm was able to find the global optimum (maximum for SMB and mimimum for Terasort and TPCx-HS), and deliver a result that was statistically very close to this optimum.

Figures 8 and 9 show a more detailed trace of how the Explorer algorithm operates. The figures show how Explorer behaved during SMB execution. Figure 8 shows container duration averages collected during all sampling intervals of one of the SMB runs. Figure 9 shows how container memory values were changed by the algorithm during this same SMB run. Although with SMB we are looking to optimize throughput, the Explorer algorithm seeks to reduce the average container duration observed during it's sampling intervals. Reduction in average container durations translates to increased throughput. During the initial explore phase Explorer cycles through all possible memory values defined in it's ranges. These cycles manifest as diagonal alignment of data points in Figure 9. Once Explorer finds the optimal container memory value, it maintains this value for all subsequent sampling intervals until it detects a change from steady state. This manifests as the horizontal alignment of data points in Figure 9. Figure 8 shows that average container durations are systematically reduced after Explorer finds the optimum container memory value.

Effects of data scale on the efficiency of the Explorer algorithm were investigated using the TPCx-HS benchmark. Table 1 shows the data collected for TPCx-HS at different data scales ranging from 300 GB to 2 TB. Data in the column

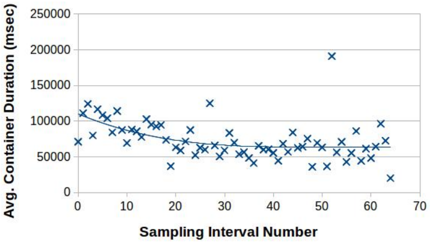

Fig. 8. Explorer algorithm optimizing container durations for SMB workload.

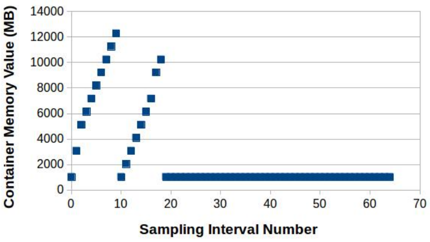

Fig. 9. Explorer algorithm optimizing container memory allocations for SMB workload.

labeled Basic Tuning shows values collected for the basic tuning baseline discussed in the previous 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 shows comparison values collected with KERMIT on-line automatic tuning. As expected KERIMT data points fall in between the two baselines.

TABLE I
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

Figure 6 shows in more detail the data points collected when establishing the best possible tuning for TPCx-HS at 300 GB data scale. For 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 were these values were changed from the default value were also conducted). 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 resulted in the slowest TPCx-HS performance. The fastest TPCx-HS performance was observed when these container memory configuration parameters were set to 3200 MB. A similar pattern was observed at all the other data scales that were tested.

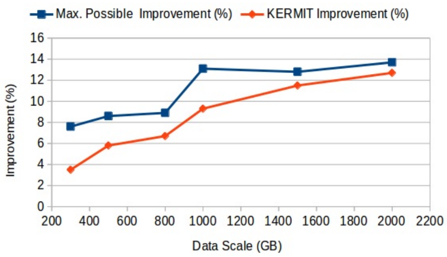

Fig. 10. Comparison of KERMIT on-line automatic tuning with the best possible tuning for TPCx-HS at different data scales.

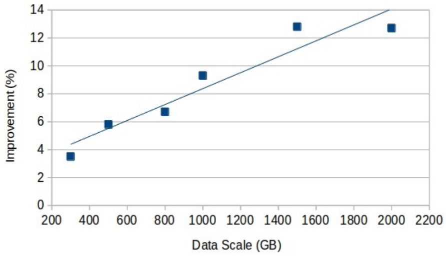

Fig. 11. Performance of KERMIT on-line automatic tuning for TPCx-HS appears to show linear increase with increasing data volume.

Figure 10 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 %) achieved relative to the basic tuning baseline discussed above. At small data scales around 300 GB, even the best possible tuning is relatively modest. However, it increases with increasing 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. We observe that KERMIT tracks the optimal tuning curve rather closely.

Figure 11 illustrates that the performance of KERMIT on-line automatic tuning for TPCx-HS shows a linear increase with increasing data volume. It is important to note that KERMIT was able to demonstrate this at much larger and more realistic data scales compared to previous efforts such as MRONLINE (2TB for KERMIT vs. 100 GB for MRONLINE), and that this improvement is measured against a tuned configuration (basic tuning) rather than the default YARN configuration used for MRONLINE. When extrapolated to larger data scales, the total performance improvement achieved with automatic tuning can be expected to continue to grow. Most actual big data deployment 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 the performance improvement to be between 20% and 30%.

Figure 12 shows the efficiency of KERMIT on-line automatic tuning as a function of data scale. Here, KERMIT efficiency is defined as the percentage of the best possible tuning performance achieved by KERMIT. For small data

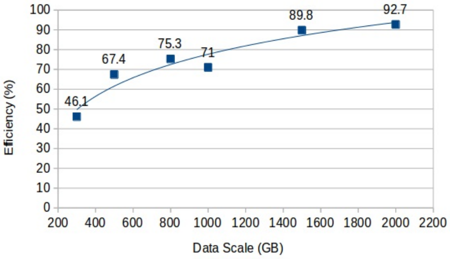

Fig. 12. KERMIT efficiency (percentage of the best possible tuning performance achieved by KERMIT) at different data scales.

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.

The main reason why KERMIT efficiency improves with increasing data scale is the fact that the total number of containers executed by YARN is directly related to the size of the data that is being processed. With larger data scale each sampling interval used by KERMIT contains more data points that are used to calculate container response time statistics. More importantly, with larger data scale jobs run longer allowing KERMIT to examine more sampling windows during the job life-cycle. In addition, with larger number of sampling windows the relative cost of Explorer searches decreases as well.

VI. Conclusion

In this paper we presented KERMIT, the first on-line automatic tuning system for YARN. KERMIT optimizes in real-time YARN memory and CPU allocations to individual YARN containers by analyzing container response-time performance. Unlike previous automatic tuning methods for specific systems such as Spark or Hadoop, this is the first study that focuses on the more general case of on-line, real-time tuning of YARN container density and how this affects performance of applications running on YARN. KERMIT employs the same tuning code to automatically tune any system that uses YARN, including both Spark and Hadoop. The effectiveness of our technique was evaluated for Hadoop and Spark jobs using the Terasort, TPCx-HS and SMB benchmarks. KERMIT on-line automatic tuning can produce configurations that are significantly better than those produced by a human tuning specialist using the shallow-tuning approach typically applied by field practitioners. In fact, KERMIT on-line automatic tuning was able to achieve performance close to the best possible parameter setting. In our case, we observed a 92.7% of the best possible performance for TPCx-HS, and close to 100% for Terasort and SMB. KERMIT configuration is very simple and does not require developers or administrators to develop complex, application-specific rules to produce significant performance improvements. Our Explorer algorithm has very low overhead and delivers the best results at larger, more realistic multi-terabyte data scales for Hadoop and for larger multi-job workloads for Spark.

Another advantage of on-line automatic tuning over manual tuning is that on-line automatic tuning can adapt in real-time to changing workload conditions, for example as map-reduce analytic jobs progress from the map to the reduce phase, or when a new job in a sequence starts, resulting in overall better and less brittle performance. It is possible to envision KERMIT working alongside application-specific automatic tuners such as MRONLINE[13], with KERMIT focusing on tuning application parameters common to all applications (such as memory, CPU, network and disk utilization allotments) and application-specific tuners focusing on tuning settings unique to the respective application.

KERMIT on-line automatic tuning running on a resource manager such as YARN dispenses with configuration settings usually left to the user, instead delegating the responsibility of finding the optimal settings to the resource manager. This will reduce the overall tuning and configuration effort required to deploy analytic applications, thereby reducing the time needed to deploy the application and the overall cost of ownership.

References

[1] 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, 2011, pp. 261–272.

[2] 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.

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

[4] 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*, ser. HPDC ’14. New York, NY, USA: ACM, 2014, pp. 253–256. [Online]. Available: http://doi.acm.org/10.1145/2600212.2600708

[5] 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*. Springer, 2009, pp. 9–20.

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

[7] 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*. IEEE, 2013, pp. 11–20.

[8] T. Lipcon, “Cloudera: 7 tips for improving mapreduce performance,” 2009.

[9] ——, “Cloudera: Optimizing mapreduce job performance,” 2012.

[10] Impetus, “Hadoop performance tuning,” 2012.

[11] 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.

[12] 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.

[13] 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*. ACM, 2014, pp. 165–176.

[14] A. S. Foundation. (2016) Apache hadoop yarn. [Online]. Available: https://hadoop.apache.org/docs/current/hadoop-yarn/hadoop-yarn-site/YARN.html