Dynamically controlling node-level parallelism in Hadoop

Kamal Kc
North Carolina State University
kkc@ncsu.edu

Vincent W. Freeh
North Carolina State University
vwfreeh@ncsu.edu

***Abstract*—Hadoop is a widely used large scale data processing framework. Applications run in Hadoop as containers, the concurrency of which affects completion time of an application as well as system resource usage. When there are too many concurrent containers, resource bottlenecks occur and when there too few, system resources are underutilized. The default and best practice settings underutilize resources which results in longer application completion times. In this work, we develop an approach to dynamically change the parallelism for concurrent containers to suit an application. Our approach ensures efficient utilization of resources and avoids bottlenecks for all types of MapReduce applications. Our approach improves performance of MapReduce applications by as much as 28% and 60% respectively when compared to the best practice and default settings.**

I. Introduction

Hadoop is an open source large scale data processing framework [1]. Programs that process large datasets can be written in MapReduce framework and deployed on Hadoop clusters [2]. To write a MapReduce program, a user only needs to implement map and reduce functions, and the framework takes care of all other operations such as creating tasks for each function, parallelizing the tasks, distributing data, and handling machine failures. YARN is an improved architecture of Hadoop and separates resource management from application logic [3], [4]. The generalization of resource management makes it easier to deploy not only MapReduce applications, but also other applications such as Spark and Tez [3].

Despite the improvement in architecture of Hadoop in YARN, performance tuning is largely manual and is done via configuration parameters. The configuration parameters control the allocation of tasks, intermediate buffer sizes, data compression, and other large number of internal operations. Achieving better system resource utilization therefore requires carefully tuning the configuration parameters, which are about 150. Previous work shows slowdown between 1.5 and 4 times due to misconfiguration [5], [6], [7]. Improving Hadoop's performance is critical due to its widespread use on a variety of applications such as indexing, log analysis, ecommerce, analytics, and machine learning [8]. In this work, we investigate and develop dynamic tuning approach for Hadoop in order to improve its resource allocation as well as performance of an application. Our tuning approach focuses on a Hadoop parameter called concurrent container slot.

Concurrent container slot (CCS) is a derived configuration parameter of Hadoop that determines how many concurrent tasks run in a node. It is derived from the total memory setting of a node and the memory setting of a container. For example, if the total memory setting of a node is 48GB and the memory setting of a container is 2GB, then the maximum number of concurrent containers that can run in the node is 24. Therefore, the CCS for the node is 24. For MapReduce applications, CCS determines how many concurrent map or reduce tasks can run at a time in a single node. Our previous work shows that an application suffers a performance degradation up to 132% when the number of concurrent tasks in a node is not appropriately configured [5]. A naïve way of determining the best CCS is to exhaustively profile the completion time for all CCS values. This approach is impractical as it is a time consuming process. Rather, there are existing best practices, some that suggest manually trying several possible values [9] and others that provide a guideline to set the configuration values [10]. Our findings show that following best practices can result in under utilization of resources and longer completion time of applications. It is therefore desirable to have an approach that effectively utilizes system resources, ensures faster completion of applications, and automatically tunes the configuration parameters. In this paper, we develop and evaluate dynamic tuning methods that to achieve these goals.

Our dynamic tuning method is based on feedback controller. We design and evaluate three different types of feedback controller. During the execution of an application, the feedback controller changes CCS in response to the change in resource usage of a system. The system resource usage is monitored at short intervals such that any bottleneck is quickly addressed. This is aided by our implementation of a suspend mechanism in YARN. When CCS is lowered to resolve the bottleneck, the extra containers are suspended, which reduces the total number of IO requests or the CPU contention. This helps to immediately get rid of bottlenecks. The efficient resource usage due to the feedback controller ensures faster completion time for applications.

There are two advantages to dynamically controlling CCS over static tuning. First, it does not require profiling. Second, the ideal CCS can change during the execution

of the application. Our findings show that our approach achieves more efficient resource usage and a performance improvement of as much as 28% and 60% respectively when compared to the best practice and the default settings. In the following sections of the paper, we present the related work, describe the design and implementation of our dynamic tuning approach, and finally evaluate the results.

II. Related work

Four areas of prior research are related to our work. The first area is Hadoop tuning. Research in this area explores tuning Hadoop using a training model [11], using tuning rules in conjunction with a hill climbing search to explore configuration parameters [12], using cost based optimization over application profile [7], using prior knowledge of optimal values of other applications [6], and automatically using resource threshold as control [13]. Our work differs from previous tuning efforts because it does not require prior knowledge, tuning rules, or extensive profiling of applications. Additionally, as our work uses dynamic control, it does not rely on predetermined resource usage threshold.

The second related area is feedback-based control. Several types of controller are used to adjust different systems. A PI controller can be used to tune web server response content quality based on its server utilization [14], and is also used to set CPU frequency of a machine by using its power consumption [15]. Another related work uses a PID controller to set CPU allocations by using machine response time [16]. Our work evaluates three different feedback controllers that use system resource usage to tune CCS.

The third related area is resource allocation. One work in this area focuses on resource sharing among multiple jobs by satisfying their dominant requested resource [17]. This ensures fairness for multiple jobs. Another work, which discusses the development of YARN, focuses on enabling each application to specify its resource requirement [3]. This helps to simplify resource allocations on each Hadoop node. In contrast, our work maximizes system resource utilization and avoids bottlenecks without specifying application resource requirements.

The fourth related area uses predefined techniques to optimize application performance. Several techniques such as rules [18], program analysis [19], or selecting alternative implementations [20] are used to optimize the performance of an application. These approaches provide an additional method of improving system performance when application usage scenarios can be enumerated before execution or when an application can be reconfigured to use alternative implementations.

III. Design and implementation

In order to dynamically change CCS, we need to know the relationship between CCS and application completion time. As completion time is not known during the execution of an application, we need a proxy for completion time that can be read continuously online. We explored many resource usage metrics to discover those that correlate with completion time. In this section, we describe these proxy metrics. We also discuss architectural changes to YARN, and design and implementation of three dynamic feedback controllers.

A. YARN architecture and containers

YARN rearchitects the original MapReduce framework by putting resource management and MapReduce application in separate modules. This enables YARN to perform only resource management tasks and makes it is useful to run other frameworks in addition to MapReduce. The resource management framework consists of a *resourcemanager* and multiple *nodemanager*s. Applications run in YARN as containers, which are process abstractions that can run any user program. These containers run in *nodemanager*s and are allocated by the *resourcemanager*. Each *nodemanager* updates its memory and virtual core limits to the *resourcemanager* [21], [22]. The memory and virtual core request of an application container is then used to determine how many containers the *resourcemanager* can allocate to a *nodemanager*.

When a MapReduce application is deployed, it creates an application master that runs as a container in a *nodemanager*. The application master asks the *resourcemanager* for containers to run map and reduce tasks. After the *resourcemanager* assigns containers to nodes, the application master coordinates their execution in conjunction with *nodemanager*s. Containers are freed after they complete map or reduce operations. Memory and virtual core limit specified in configuration files determine the number of concurrent containers that run in a node. Misconfiguration may result in fewer or greater concurrent containers slots (CCS) than ideal. This may result in slowdown of a MapReduce application [5].

Our implementation of dynamic CCS uses a distributed architecture where each *nodemanager* periodically computes CCS. This new CCS value is propagated to the *resourcemanager* using protocol buffer messaging API. The existing YARN *resourcemanager* uses memory values to determine a node's CCS, but the *resourcemanager* in our implementation directly uses the computed CCS value to allocate containers. The *nodemanager* ensures that the memory requirement of all the running containers does not exceed the available physical memory of the node.

B. Dynamic CCS

The operations performed by a *nodemanager* to dynamically change CCS are as follows. In the *nodemanager*, a thread samples the system resource usage periodically. The selected resource metrics are indicators of good and/or bad performance. These metrics and their relationship with

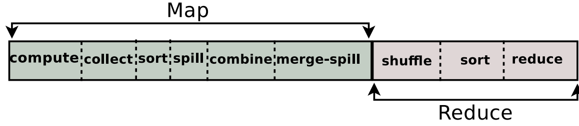

Figure 1: Map and reduce task phases.

performance are described in the Section III-C. A feedback controller uses these metric values to compute the CCS for next interval. If the new CCS is lower than the current CCS, then the extra container processes are suspended. If the new CCS is higher and there are existing suspended containers, then they resume execution. The suspended containers resume before any new containers spawn. This happens even when the CCS remains the same and the running containers complete execution. When there are no suspended containers and the current number is fewer the current CCS, then newer containers are assigned by the *resourcemanager*.

YARN does not support suspending containers. The mechanism to suspend containers was implemented as a part of this work. Containers are suspended by using the operating system signal mechanism. The signal SIGSTOP suspends a container process and the signal SIGCONT resumes it [23]. The suspend implementation keeps track of each container in one of the two queues – run queue and suspend queue. Similar suspend mechanism implementation is used in a previous work [24].

The purpose of decreasing CCS is to avoid instantaneous and transient IO or CPU bottlenecks. Therefore, it is imperative to reduce parallelism (by suspending containers) immediately. A secondary advantage of suspend mechanism is that the system resource usage can be measured in shorter intervals. As new CCS takes effect immediately, a shorter interval such as 10 seconds is sufficient. This measurement interval is small enough to have quicker response and large enough to generate sufficient container activity and exhibit negligible overhead. However, further investigation is needed to determine ideal measurement interval.

C. Resource usage monitoring

In order to determine the types of resource metrics that are relevant to tuning a MapReduce application, we describe map task resource usage in detail. A map task consists of six phases. These phases as shown in Figure 1 are compute, collect, sort, spill, combine, and merge-spill. In the compute phase, a map task performs the map function computation on each input key-value pair. In the collect phase, it stores the processed key-value pairs in an output buffer. The output is logically divided into partitions equal to the total number of reduce tasks. In the sort phase, the map task sorts the output key-value pairs of each partition. When the output buffer is full, the map task spills its content to a spill file in the local disk. This is the spill phase. The combine phase is optional and when present, the map task performs a local reduce operation on the output key-value pairs. At the end, if there are multiple spill files then they are merged together to produce a single map output file. This is the merge-spill phase. Each map task phase is either CPU or IO intensive. The CPU intensive phases are compute, collect, sort, and combine. The IO intensive phases are spill and merge-spill.

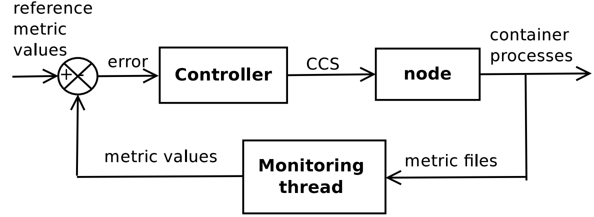

Figure 3: Feedback control for dynamically changing CCS.

Performance metrics of a Hadoop application can be collected from two different sources – Hadoop counters and operating system statistics. Hadoop counters generate statistics about container tasks, jobs, and HDFS operations. Operating system statistics are more general and include information on the CPU utilization, IO read and writes, context switches, blocked processes, and other system statistics [25]. In this work, we use operating system statistics because it provides instantaneous information on the state of a system and can be used to quickly identify if the system has degraded due to resource bottlenecks. As our work uses Linux, these statistics are available in the files `/proc/stat` and `/proc/diskstats` [25].

Ideally, a metric can be used to determine application performance if it can be distinctly mapped to a performance value. No operating system metrics reliably predict completion time on there own [26]. Three metrics shown in Figure 2 are most helpful in separating good and bad performance. Each point in the figures represents a 30-second average of the metric values. The figures contain measurements for six diverse applications. The map completion time for each point is for the application run in which the sample measurement is taken, which explains the vertical grouping of points. As the y-axis is completion time, lower is better.

The metric shown in Figure 2a is *user_cpu*, which is the percentage of time spent by CPU in user mode, a high value indicates good performance. The metric shown in Figure 2b is the number of processes blocked on IO, a high value indicates poor performance. The metric shown in Figure 2c is the total number of context switches, a high value indicates good performance. However, a small range in the high value results in bad performance for context switches. These three metrics form the basis of measuring resource usage in our work.

D. Feedback controllers

Feedback controllers change the state of a system based on its current state. In our implementation, a feedback controller works as shown in Figure 3. The input to the controller is error, which is the difference between the

(a) (b) (c)

Figure 2: Effect of user CPU utilization, blocked processes, and context switches in completion time.

measured resource metric values and the reference metric values. Reference metric values are the desired values the system targets to achieve the shortest completion time. One controller in our work uses fixed metric values as reference values, whereas others do not and and rely on other comparable reference values. Nevertheless, each controller has input that is an error value representing the deviation of current system state from the desired one. Based on this error value, the controller computes the next CCS and the *nodemanager* makes sure that CCS numbers of container processes are running in the system. The goal of the controller is to minimize the error, ideally to zero. In this work we evaluate three different types of controllers as described below.

1) Waterlevel controller: Waterlevel controller mimics a water tank controller which uses two water level markers — a lower mark and an upper mark, to maintain water content in the tank. When water level decreases below the lower mark, water starts filling the tank and when the level reaches the upper mark, it stops. This process continues and the controller always keeps water level between the two marks.

An equivalent Waterlevel controller can be constructed for dynamic CCS by choosing the lower and upper marks as threshold values for the three metrics – *user_cpu*, *procs_blocked*, and *ctxt*. This controller works as follows. When any metric decreases below the lower mark, CCS increases[1]. On the other hand, when any metric value increases above the upper mark, CCS decreases. Increasing and decreasing CCS also respectively increases and decreases the metrics, except for some cases discussed in the previous Section III-C. The error input, as shown in the Figure 3, therefore has one of the two values – 0 or 1, 0 when a metric value is smaller than the lower mark and 1 when a metric value is greater than the upper mark. The controller increases CCS when the error is 0 and decreases it when the error is 1. The controller increases or decreases CCS by a fixed amount (e.g., 2). In order to ensure that the resource usage is similar to the upper mark, the two lower and upper marks are set closer. This ensures efficiency in resource usage.

2) PD controller: A PD controller[2] has proportional and derivative components that affect its output [27], [28]. Equation 1 shows how the next CCS is calculated using the coefficients, Kp (proportional) and Kd (derivative), that weigh the error value E(k) measured at k^{th} time. Kp uses current error and Kd uses change in the error.

$$\Delta CCS = K_p E(k) + K_d (E(k) - E(k-1)) \tag{1}$$

Unlike the Waterlevel controller, the PD controller generates new CCS rather than a binary increase or decrease value. To compute a single error value from the three metrics, *user_cpu*, *procs_blocked*, and *ctxt*, we need to combine them together. As these metrics have different value ranges and different correlation with application time, we combine the metrics to form a parameter called score that accounts for these factors. Space does not permit us to adequately explain the score function. However, the score was designed so that a higher score represents a good performance whereas a lower score represents bad performance.

As described earlier, a PD controller relies on the error to derive new CCS values. If the maximum achievable score of an application is known, then the error parameter is the difference between the maximum score and the measured score. But, the maximum score is not known before executing an application, because applications have different resource usages. Therefore, we construct an error parameter that becomes 0 when the score is maximum, becomes positive if CCS needs to increase, and becomes negative if CCS needs to decrease. This error parameter is the rate of change of score compared to the change in CCS. It is shown in

[1]Metrics can disagree (i.e., one is high and one low). Our solution takes this into account. However, these metrics are correlated, this case has never been observed.

[2]We experimented with a PID controller. But the integral component (I) is not appropriate because it accumulates error, which does not apply in this case.

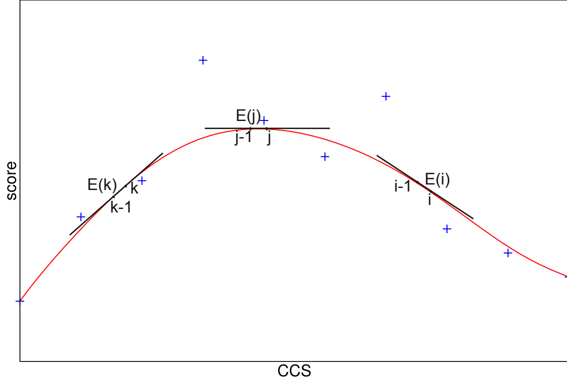

Figure 4: Bezier approximation of score against CCS.

Equation 2 as $E(k)$, where k represents the measurement for k^{th} time sample.

$$E(k) = \frac{score(k) - score(k-1)}{CCS(k) - CCS(k-1)} \tag{2}$$

E(k) is the slope of the curve between score and CCS. Figure 4 shows the Bezier approximation of score as a function of CCS for terasort. As mentioned previously, the score is measured empirically using the three metrics and indicates good performance when high and bad performance when low. The plot shows that as CCS increases, the score initially increases, peaks and then gradually decreases. This behavior is similar for other applications. In the figure, at point j, the score is the highest and the error $E(j)$ is 0. In this case, the controller attempts to maintain the same CCS. At point k, the score is increasing and the error $E(k)$ is positive. In this case, the controller increases CCS. At point i, the score is decreasing and the error $E(i)$ is negative. In this case, the controller decreases CCS.

The gain parameters are tuned one at a time. Initially, Kp is tuned and then Kd. The main goals of tuning are to obtain parameter values that have quick ramp up time and do not result in large CCS changes. For this, each parameter is assigned a set of values in decreasing order and resulting CCS change is observed. For Kp the values used were 4.0, 2.0, 1.0, and 0.5. With 4.0, there were large changes in CCS values, but as Kp decreased, the changes also decreased. The value of 1.0 was selected for Kp because with 0.5 even though the CCS changes decreased, the ramp up was slow. Next, Kd is tuned in similar manner by keeping Kp constant at 1.0. For Kd, the values used were 2.0, 1.0, 0.5, and 0.1. The selected value that best fit both the goals was 0.5. Therefore, this work uses Kp and Kd values of 1.0 and 0.5.

3) PD+pruning controller: Pruning can be done together with a PD controller to eliminate changes in CCS after the controller finds the one with the best score. CCS tuned by due to PD controller frequently change, as instantaneous resource usage changes quite often even for same CCS. PD+pruning addresses this behavior and eliminates the constant changing of CCS. While tuning, if a new CCS results in a higher score, then old CCS is pruned. When the new CCS does not result in a higher score, then the controller attempts to find higher score for CCS in the range between the new and the old CCS. If it finds any such CCS, it continues with the tuning. Otherwise, the highest score the controller could find is one with the old CCS. This CCS is then used through the remainder of application execution.

Applications	CPU_UTIL (%)	IO_THRPUT (MB/s)	Application type
terasort	42	10.58	*IO-intensive*
rankedinvertedindex	50	7.93	
word count	63	4.97	*Balanced*
invertedindex	72	5.45	
termvectorperhost	74	5.53	
grep	99	0.01	*CPU-intensive*

Table I: Types of applications based on CPU and IO characteristics.

IV. Evaluation

A. Experimental setup

The cluster used in this work consists of ten IBM PowerPC machines. Each node has two POWER7 processors with 16 cores – 64 total CPU threads, 124GB RAM, and a 10 Gbps Ethernet network link. Hadoop (version 2.2.0) is configured with one *resourcemanager* and nine *nodemanager*s. HDFS is configured with one *namenode* and nine *datanode*s.

Our experiments use six Hadoop applications from the PUMA benchmark [29]. These six applications, as shown in Table I, were selected to diversify the applications types based on their per map task CPU utilization (CPU_UTIL) and write IO demand (IO_THRPUT). CPU_UTIL is the percentage of total map task time that is spent on the CPU intensive phases of a map task. IO_THRPUT is the rate (in MB/s) at which a map task writes data to its output files. Having applications with all ranges of CPU_UTIL (42% to 99%) and IO_THRPUT (10.58 MB/s to 0.01 MB/s) ensures that our results are applicable to all types of Hadoop applications. In our previous work, we found that using CPU_UTIL values applications can be grouped into three regions [5]. *IO-intensive* applications have high IO and low CPU usage, *Balanced* have medium IO and medium CPU usage, and *CPU-intensive* have low IO and high CPU usage. The best suitable CCS values are different for each of these regions. Except for *terasort*, all other applications use wikipedia dataset available in PUMA benchmark. The dataset size is 900GB, which is formed by combining multiple copies of the original 300GB wikipedia PUMA dataset. *Terasort* uses the data generated by teragen program [1].

Apart from the three dynamic approaches, we also compare other alternative tuning techniques. They are described as follows.

Applications	Default	Best practice	Optimal	WL	PD	PD+ pruning
terasort	1.57	**1 (1560)**	0.89	0.93	0.90	0.91
rankedinvertedindex	1.58	**1 (2102)**	0.79	0.83	0.79	0.80
wordcount	1.68	**1 (2521)**	0.73	0.78	0.73	0.76
invertedindex	1.66	**1 (2976)**	0.77	0.81	0.77	0.80
termvectorperhost	1.69	**1 (3272)**	0.79	0.80	0.79	0.80
grep	1.82	**1 (2526)**	0.68	0.70	0.72	0.69

Table II: Relative comparative map completion times for various CCS settings. The best practice column shows total elapsed time in seconds.

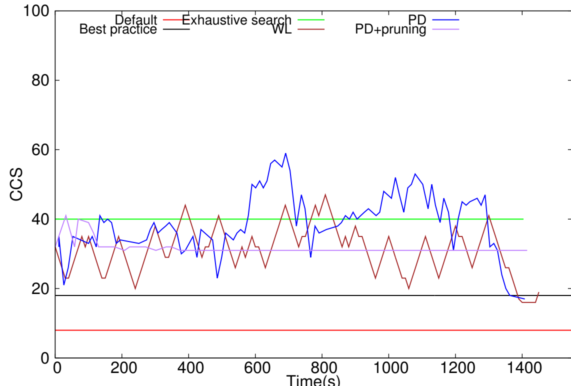

Figure 5: CCS changing for terasort for all tuning approaches.

1) Default values: Hadoop distribution has default configurations for YARN as well as for MapReduce. The default configuration is 8 concurrent map tasks or 4 concurrent reduce tasks.

2) Best practice: Guidelines for best practice suggest setting aside specific amounts of CPU and memory resource for the system and background operation [10], [30], [31]. Best practices suggest a CCS value based on available resources, specifically number of cores, number of disks, and amount of memory. Best practices recommend CCS be 1.5 times the number of physical cores (24 in our case). It suggests a CCS equal to 1.8 times the number of disks (18 in our case). Finally, it suggests reserving some memory for the system and allocating the remainder to containers. In our case, the system reserve is 24GB, so 104GB divided by 2GB per container suggests CCS of 52. Further best practices suggests taking the minimum of these three values. Therefore, the best practice CCS used in this paper is 18.

3) Exhaustive search: Exhaustively searching all possible CCS values can also find the optimal CCS value of an application. However, this approach has two disadvantages – it takes a long time to exhaustively search the best CCS and the search has to be performed for each application.

B. Performance comparison

Table II shows relative comparison between map completion times for various CCS tuning approaches. The comparison is relative to the time for best practice. The table shows

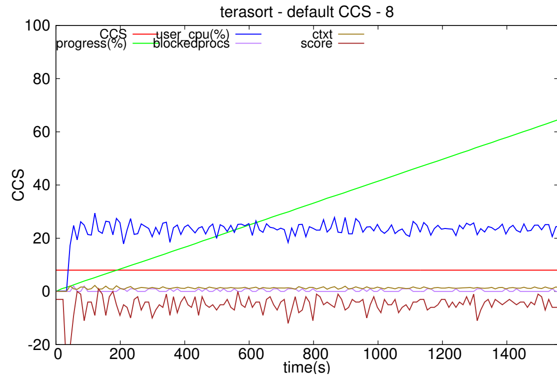

Figure 6: Resource usage of a *nodemanager* during map execution of *terasort* for default CCS.

that the default configuration is at least 50% slower for all applications. All three dynamic approaches are much better than best practices (7-31% better). All are within 4% of exhaustive (ie, optimal). While all 3 are good, PD is slightly better. It is on average 21% better than best practice and 1% worse than optimal. Among the dynamic approaches, PD is better than WaterLevel (WL) and PD+pruning for all applications except for *grep*, where it is slower by 1% and 2% respectively. Compared to the exhaustive search, PD controller is equally better for four applications and slower for the remaining two by a maximum of 4%.

An effective tuning approach needs to change CCS to address resource usage fluctuations. When an application runs, the CPU and IO operations performed by map tasks interleave causing changes in resource usage of a node. Furthermore, resource usage fluctuations also occur due to different types of tasks, for example, reduce tasks have different resource usage characteristics than map tasks. Figure 5 shows the CCS values over time on one node during the execution of terasort, in order to illustrate the difference in how the approaches respond to system resource usage. The default, best case, and exhaustive search are static approaches, and therefore do not change CCS. Among the dynamic approaches, PD and WL changes CCS. PD+pruning changes CCS initially, but stabilizes CCS to a fixed value after the 350 second mark. This means that it does not respond to any resource usage change after 350 seconds. Table II and Figure 5 therefore show that among all approaches, dynamic tuning using PD controller achieves the most satisfactory performance as well as CCS responsiveness.

C. Resource usage

Figure 6 shows *nodemanager* CCS, resource usage (user CPU, blocked processes, and context switches), score, aplication progress of *terasort* with default CCS of 8. The figure shows highest CPU usage at 30%, low blocked_procs and ctxt value of 2, and score close to 0. In this case, the resource is underutilized, as there is sufficient spare CPU resource

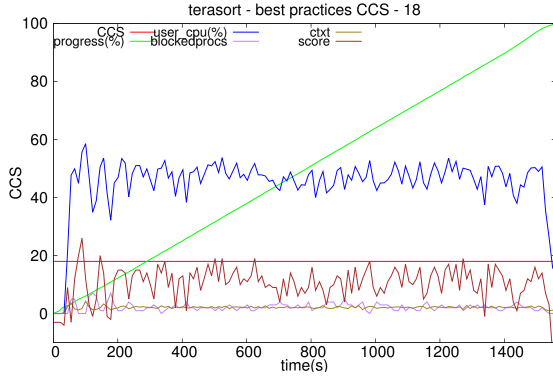

Figure 7: Resource usage of a *nodemanager* during map execution of *terasort* for best practices CCS.

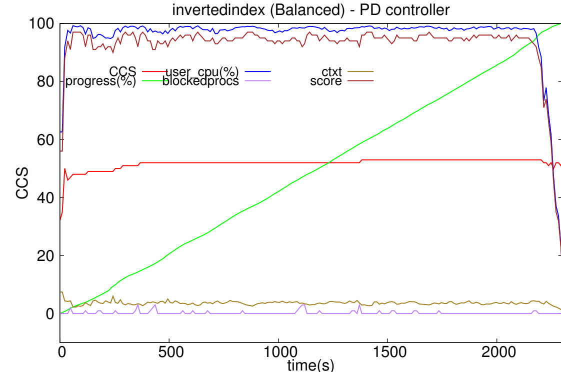

Figure 9: Resource usage of a *nodemanager* during map execution of a Balanced application (*invertedindex*) for PD.

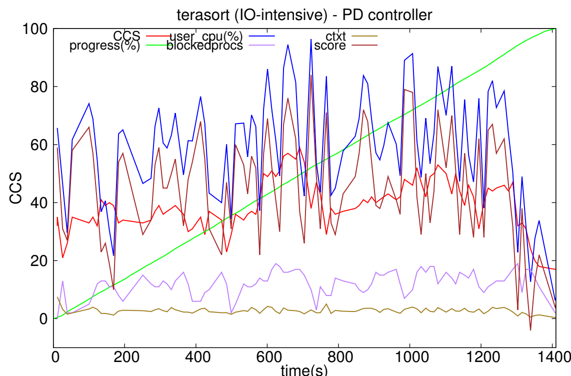

Figure 8: Resource usage of a *nodemanager* during map execution of an IO-intensive application (*terasort*) for PD.

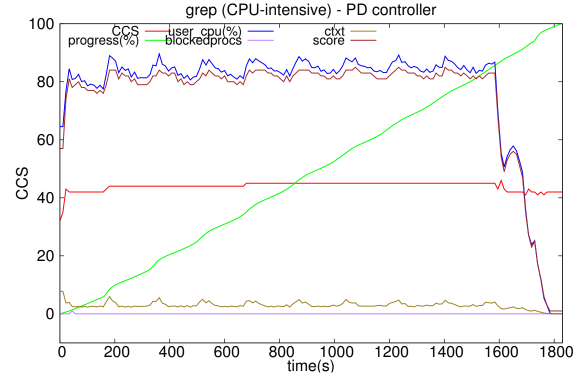

Figure 10: Resource usage of a *nodemanager* during map execution of a CPU-intensive application (*grep*) for PD.

available. Figure 7 shows the resource usage, score, and progress for *terasort* with best practice CCS of 18. In this case also, the CPU resource is underutilized. The CPU usage peaks at 60%, and low blocked_procs and ctxt have values of 2.

Figures 8, 9, and 10 show the resource usages, score, and application progress for IO-intensive (*terasort*), *Balanced* (*invertedindex*) and *CPU-intensive* (*grep*) applications. All three figures show CPU resource being appropriately used. For IO-intensive application in Figure 8, the CPU usage has a peak value of 90%. It is relatively lower than the other two because it has a higher blocked_procs. Whenever CCS increases, CPU usage increases, but blocked_procs increases as well. This results in lower score and indicates increasing IO bottleneck. Therefore, the controller lowers CCS. For *Balanced* application in Figure 9, CPU utilization increases but blocked_procs and ctxt remain low. Therefore, CPU usage is as high as 99%. For *CPU-intensive* application in Figure 10, with increasing CCS, CPU utilization increases but ctxt increases as well. Hence, the controller adjusts CCS and does not allow ctxt to increase significantly. The resulting CPU usage peaks at 90%. These three figures show that the dynamic approach utilizes available resources more efficiently than the best practice and default.

D. Tuning CCS for multiple workloads

During the execution of an application, the best CCS can change due to resource pressure caused by other workloads running in the system. To demonstrate this, we run a background IO bound workload while *terasort* is executing. This background workload generates write traffic with a maximum of 32 *procs_blocked*. Figure 11 shows the result. Initially, the PD controller tunes CCS to values around 40. At 460 second, the background workload starts and runs until 980 second mark. During this time, the *procs_blocked* value is high, and the resulting score is low. Therefore, the controller decreases the CCS. After the workload completes execution, the CCS increases back to value above 40. It completes in 1760 seconds, with best practice CCS and same background workload it takes 2000 seconds. Dynamic approach has a improved performance of 12% over best practice, which is 2% higher than when comparing them without any background workload. Therefore, this experiment illustrates the effectiveness of PD controller in tuning CCS even when there are other workloads in a system.

Summary. The resource usage for the static and dynamic approaches showed that having a dynamic tuning method

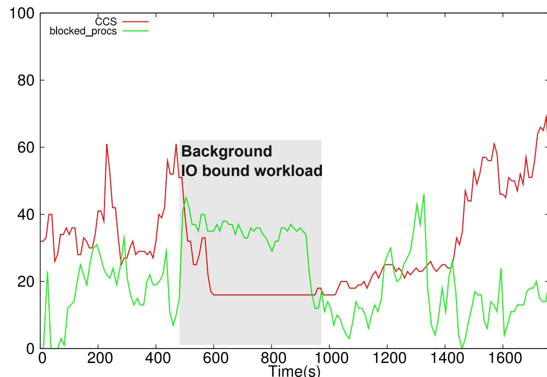

Figure 11: Dynamic CCS reponse to background IO workload for *terasort*.

adjusts appropriately to resource usage fluctuations, does not under utilize resources, and avoids resource bottlenecks. Additionally, the performance comparison showed improvement up to 28% for PD based dynamic CCS when compared to the best practice and 60% when compared to default.

V. Conclusion

In this paper, we explored feedback based dynamic approaches to tune concurrent container slot (CCS) and improve performance of Hadoop applications. Our dynamic approaches use controllers that take instantaneous score or score to CCS ratio as input and generate new CCS as output. The score is a combination of user CPU, blocked processes, and context switches values. We evaluated WaterLevel, PD, and PD+pruning controllers. While all dynamic controllers performed better than the best practice, PD controller was comparably better with highest performance benefit of 28% as well as with better CCS responsiveness to fluctuations in resource usage. The result is even better with 60% improvement compared to default settings. In order to ensure that the findings are applicable to all types of MapReduce applications, the six applications selected in this work have diverse CPU and IO usage profiles. In conclusion, our findings suggest that for Hadoop applications, instead of using existing best practice and default settings, using dynamic tuning offers better performance and system responsiveness while avoiding resource bottlenecks.

References

[1] "Hadoop," http://hadoop.apache.org.

[2] J. Dean and S. Ghemawat, "Mapreduce: Simplified data processing on large clusters," in *OSDI*, 2004.

[3] V. K. Vavilapalli, A. C. Murthy, C. Douglas, S. Agarwal, M. Konar, R. Evans, T. Graves, J. Lowe, H. Shah, S. Seth, B. Saha, C. Curino, O. O'Malley, S. Radia, B. Reed, and E. Baldeschwieler, "Apache Hadoop YARN: Yet Another Resource Negotiator," in *SOCC*, 2013.

[4] "Apache Hadoop Next Generation MapReduce (YARN)," http://hadoop.apache.org/docs/r2.4.0/hadoop-yarn/hadoop-yarn-site/YARN.html.

[5] K. Kc and V. W. Freeh, "Tuning Hadoop map slot value using CPU metric," in *BPOE*, 2014.

[6] K. Kambatla, A. Pathak, and H. Pucha, "Towards optimizing hadoop provisioning in the cloud," in *HotCloud*, 2009.

[7] 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*, 2011.

[8] "Hadoop poweredby," http://wiki.apache.org/hadoop/PoweredBy.

[9] "Avoiding common hadoop administration issues," http://blog.cloudera.com/blog/2010/08/avoiding-common-hadoop-administration-issues.

[10] "Installing HDP Manually," Hortonworks Data Platform v 2.2, Hortonworks (Page 30).

[11] Z. Zhang, L. Cherkasova, and B. T. Loo, "AutoTune: Optimizing Execution Concurrency and Resource Usage in MapReduce Workflows," in *ICAC*, 2013.

[12] M. Li, L. Zeng, S. Meng, J. Tan, L. Zhang, A. R. Butt, and N. Fuller, "MRONLINE: MapReduce Online Performance Tuning," in *HPDC*, 2014.

[13] K. Wang, B. Tan, J. Shi, and B. Yang, "Automatic Task Slots Assignment in Hadoop MapReduce," in *ASBD*, 2011.

[14] T. Abdelzaher and N. Bhatti, "Web server QoS management by adaptive content delivery," in *IWQoS*, 1999.

[15] R. J. Minerick, V. W. Freeh, and P. M. Kogge, "Dynamic Power Management using Feedback," in *COLP*, 2002.

[16] X. Wang and Y. Wang, "Coordinating Power Control and Performance Management for Virtualized Server Clusters," *IEEE TPDS,*, 2011.

[17] A. Ghodsi, M. Zaharia, B. Hindman, A. Konwinski, S. Shenker, and I. Stoica, "Dominant Resource Fairness: Fair Allocation of Multiple Resource Types," in *NSDI*, 2011.

[18] "Hadoop vaidya," http://hadoop.apache.org/docs/stable/vaidya.html.

[19] C. Olston, B. Reed, A. Silberstein, and U. Srivastava, "Automatic optimization of parallel dataflow programs," in *USENIX ATC*, 2008.

[20] C. Whaley, A. Petitet, and J. J. Dongarra, "Automated Empirical Optimization of Software and the ATLAS Project," *Parallel Computing*, 2000.

[21] "JIRA YARN-1024 Define a CPU resource(s) unambiguously," https://issues.apache.org/jira/browse/YARN-1024.

[22] "JIRA YARN-1089 Add YARN compute units alongside virtual cores," https://issues.apache.org/jira/browse/YARN-1089.

[23] "SIGNALS(7) - Linux Programmer's manual," http://man7.org/linux/man-pages/man7/signal.7.html.

[24] M. D. Mario Pastorelli and P. Michiardi., "OS-Assisted Task Preemption for Hadoop Jobs," in *DCPerf*, 2014.

[25] "PROC(5) - Linux Programmer's manual," http://man7.org/linux/man-pages/man5/proc.5.html.

[26] K. Kc and V. W. Freeh, "Dynamic performance tuning of Hadoop," *Technical Report, North Carolina State University*, 2014.

[27] D. E. Seborg, D. A. Mellichamp, T. F. Edgar, and F. J. D. III, *Process Dynamics and Control*. Wiley.

[28] J. L. Hellerstein, Y. Diao, S. Parekh, and D. M. Tilbury, *Feedback Control of Computing Systems*.

[29] F. Ahmad, S. Lee, M. Thottethodi, and T. N. Vijaykumar, "Puma: Purdue mapreduce benchmarks suite," *TR, Purdue University*, 2012.

[30] "Migrating to MapReduce 2 on YARN," http://blog.cloudera.com/blog/2013/11/migrating-to-mapreduce-2-on-yarn-for-operators/.

[31] E. Sammer, *Hadoop Operations (Page 124)*.