Evaluation of MapReduce in a large cluster

Kamal Kc, Chin-Jung Hsu, and Vincent W. Freeh

{kkc,chsu6,vwfreeh}@ncsu.edu
North Carolina State University

***Abstract*—MapReduce is a widely used framework that runs large scale data processing applications. However, there are very few systematic studies of MapReduce on large clusters and thus there is a lack of reference for expected behavior or issues while running applications in a large cluster. This paper describes our findings of running applications on Pivotal's Analytics Workbench, which consists of a 540-node Hadoop cluster. Our experience sheds light on how applications behave in a large-scale cluster. This paper discusses our experiences in three areas. The first describes scaling behavior of applications as the dataset size increases. The second discusses the appropriate settings for parallelism and overlap of map and reduce tasks. The third area discusses general observations. These areas have not been reported or studied previously. Our findings show that IO-intensive applications do not scale as data size increases and MapReduce applications require different amounts of parallelism and overlap to minimize completion time. Additionally, our observations also highlight the need for appropriate memory allocation for a MapReduce component and the importance of decreasing log file size.**

I. Introduction

Hadoop is a popular open source implmentation of MapReduce framework [1]. MapReduce framework runs large scale data processing applications on a cluster [2]. To write a MapReduce application, a programmer only needs to implement map and reduce functions, and the Hadoop MapReduce framework manages all other operations such as creating tasks for each function, parallelizing the tasks, distributing data, and handling machine failures. Various types of applications such as indexing, log analysis, ecommerce, analytics, and machine learning run on Hadoop [3]. Our work evaluates Hadoop on Pivotal's Analytics Workbench (AWB), which contains a 540-node Hadoop cluster [4].

Although there are multiple known deployments of Hadoop clusters greater than 500 nodes, there is little published work about MapReduce on a large cluster [3]. There is a study analyzing the logs of 6000 node cluster that highlights the need to make resource allocation aware of diversity in node resources [5]. Another study on logs of a 3000 node cluster examines the diversity of applications running on the cluster [6]. However, these studies do not identify the expected behavior of applications when running in a large cluster.

This paper studies three facets of MapReduce applications in a large cluster. Our first evaluation explores data scaling behavior of six MapReduce applications that have wide variety in map and reduce task characteristics. Our second evaluation explores the effect of configuration parameters on application performance. Previous studies on small clusters (10 and 14 nodes) have shown that tuning task parallelism improves overall performance [7], [8]. However, the effects of changing parallelism and overlap of tasks in MapReduce applications in a larger cluster have not been studied. Our third evaluation discusses experiences of running applications in a large cluster. This includes findings that are not noticeable in smaller clusters.

Our results on data scaling show that for applications that require higher IO throughput, the completion time increases in higher proportion than the increase in data size. On the other hand, for applications that require medium to low IO, the completion time increases proportionally to the increase in data size. Our results on parallelism and overlap configuration show that an appropriate configuration of parallelism and overlap improves the performance of an application. This configuration value is dependent upon application CPU and IO characteristics. Our findings on AWB use shows observations that are not noticeable in a smaller cluster. We discuss two such observations in this paper — appropriately sizing a component of an application and the suitability of the type of information level for huge application logs produced by MapReduce applications.

The remainder of the paper is organized as follows. Section II presents a background on Hadoop and related work. Section III details our experiment methodology. Section IV describes our findings on data scaling behavior of MapReduce applications. Section V describes our findings no tuning parallelism and overlap configuration of an application. Section VI describes our experiences on observations for a large cluster.

II. Background

This section describes Hadoop and prior work.

A. Hadoop and MapReduce

The Hadoop architecture consists of a MapReduce computing framework [2] and a data storage framework called Hadoop Distributed File System (HDFS) [9]. HDFS is formed by aggregating local storage of each client node.

It stores data in 64MB or larger block sized files. This work uses the rearchitected Hadoop framework called YARN [10], [11]. Unlike the original MapReduce, YARN separates resource management from the MapReduce application logic. The resource management framework consists of a single *resourcemanager* and multiple *nodemanagers*. The resource management framework is generalized such that it can run MapReduce as well as other applications. The container is the unit of resource allocation and applications run in these containers. Each *nodemanager* informs to the *resourcemanager* of its memory and CPU capacity. Using each *nodemanager*'s resource capacity and the container memory resource request of an application, the *resourcemanager* determines the *nodemanager* in which the container runs.

When a MapReduce application is deployed, it creates an application master (AM) that runs in a container. The AM negotiates containers with *resourcemanager* and coordinates the execution of map and reduce tasks in the allocated containers. A map task processes a block of HDFS data. The map task is typically assigned to a container local to the node which has the input data block. The input to a map task is in the form of key-value pairs. The map task partitions its output by keys. The total number of partitions is equal to the number of reduce tasks. The output is then stored in the local disk. A reduce task processes one specific partition of the map output. The data processed by a reduce task is the aggregated collection of its assigned partition that is copied from the outputs of all map tasks. Reduce tasks write their output, which is the final output, to HDFS.

B. Parallelism and overlap configuration

Configuration parameters in YARN affect resource allocations, scheduling, and logging. This work also attempts to characterize the performance effects of parallelism and overlap in MapReduce applications. Two configuration parameters — concurrent container slot and slowstart determine the parallelism and overlap respectively.

Concurrent container slot (CCS) determines the parallelism of MapReduce tasks. It is the maximum number of concurrent containers that can run in a single node. It is a derived configuration parameter and is equal to the total memory configured for a *nodemanager* divided by the memory request for each container. The *nodemanager*'s configured memory is shared among map and reduce containers. Reduce containers usually have bigger memory size as a typical reduce task processes more data than a typical map task. The default memory configuration for a *nodemanager* is 8GB, for a map container is 1GB, and for a reduce container is 2GB. This results in CCS of 8 when only map tasks run. But, when reduce tasks also run the node, the total number of concurrent containers will be less than 8.

Slowstart (SS) determines the overlap between map and reduce tasks. Its value is equal to the percentage of map tasks that complete before reduce tasks start. A SS value of 75% means that the reduce tasks start as soon as 75% of the total map tasks complete. The default value of SS is 5%. Not all reduce tasks are immediately scheduled. During reduce scheduling, the AM ensures that sufficient containers are available to run map tasks. Once a reduce task starts, it runs to completion, which is necessarily after all map tasks are complete. Therefore, a reduce task, once started, will occupy its container until the end of the entire application. Consequently, starting reduce tasks too early (small SS) can result in underutilization when reduce tasks wait for map output. On the other hand, starting too late (large SS) can result in delayed copy of map output and resource burden to store the intermediate map output. Therefore, slowstart presents a trade-off. Section V discusses in detail the effects of changing CCS and SS values for different types of applications in AWB.

C. Related work

There are three areas of prior research that are related to our work. The first area is large scale cluster study. The original mapreduce work describes applications running in a large 1400-node cluster [2]. There are known multiple large Hadoop clusters with more than 500 nodes [3]. Prior studies on large Hadoop clusters primarily focus on analysis of workloads using log traces and addressing Hadoop runtime issues. A study of log traces of clusters containing upto 3000 nodes showed that applications in large clusters have different input, shuffle data, output, completion time, and submission pattern [6]. Another study of logs of cluster with up to 6000 nodes showed that resource allocation in Hadoop should address the differences in per node CPU and memory capacities and mixing of short and log Hadoop jobs [5]. Another previous work showed that jobs were failing in a large Hadoop cluster due to incorrect configuration of the storage framework [12].

The second related area is scalability. Previous work compares scale-out and scale-up models and investigates the relative effectiveness of scale-up [13]. Similarly, another related work compares the performance of scale-out and scale-up in Hadoop and finds performance improvement with the scale-up model [14]. Another work studies the scalability of an indexing application in Hadoop when changing the cluster size [15]. Another related work uses IO model and data parsing to address performance bottlenecks in Hadoop [16]. Previous work also explores the limitation of Hadoop scalability in terms of its inability to change map and reduce slots for different applications [17].

The third related area is benchmark study. PUMA provides a collection of applications for MapReduce benchmarking [18]. The applications used in this work are from PUMA benchmark. HiBench provides both microbenchmark and real-world benchmark for Hadoop [19]. YCSB is a cloud benchmark that attempts to understand the performance and behavior of cloud data serving systems

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: Three types of applications in PUMA benchmark.

Applications	Input	Map output	Shuffle	Reduce output
terasort	17.60 TB	17.93 TB	18.29 TB	17.58 TB
rankedinvertedindex	16.70 TB	17.37 TB	17.83 TB	17.13 TB
word count	19.40 TB	23.55 TB	3.62 TB	5.15 GB
invertedindex	19.40 TB	29.54 TB	3.98 TB	43.85 GB
termvectorperhost	19.40 TB	31.92 TB	4.13 TB	46.35 GB
grep	19.40 TB	1.39 GB	4.20 GB	457 KB

Table II: Comparative data movement through the MapReduce phases for Medium dataset.

[20]. TPC-W is a benchmark to test the scalability of e-commerce websites [21].

III. Experimental methodology

In order to make the findings of this work applicable to all types of applications, we select them based on the diversity of their CPU and IO activity. We use six out of the thirteen benchmark programs in PUMA [18]. The programs are *grep*, *word count*, *invertedindex*, *rankedinvertedindex*, *terasort*, and *termvectorperhost*. The CPU usage of these six programs range from a low value to a very high value. Similar diversity exists for IO throughput. Remaining PUMA applications similar CPU and IO characteristics.

The CPU and IO usage diversity is observable when measuring the time spent by a map task in CPU intensive phases of a map task. Table I lists the CPU and IO usage values for a single map task. 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 at which a map task writes data to its output files and it is measured in megabytes per second. The values suggests distinction among the CPU and IO usage of the applications. CPU_UTIL varies from 42% to 99% and IO_THRPUT varies from 10.58 MB/s to 0.01 MB/s. Based on the range of low, medium, and high CPU_UTIL values shown in Table I, the applications are categorized into three regions: *IO-intensive*, *Balanced*, and *CPU-intensive*. Our previous study shows that these categories have different map task paralellism that performs best [8].

The diversity in the CPU and IO usage of map tasks also affects the reduce task behavior because the size of the reduce input is dependent on the amount of write IO activity performed by maps. The map output data is processed by a combiner, which performs local reduce of the map output data and produces the shuffle data. Table II shows the size of data that moves across these phases. An important observation is the difference in the data size for shuffle operation for applications of different regions. Shuffle data is copied over the network and serves as input for reduce tasks. Therefore, diversity in the reduce input size affects the time spent by reduce tasks in network copy, merge, sort, and reduce operations. The *IO-intensive* applications have shuffle data as large as the input data size and therefore require higher network throughput during data copy. The *Balanced* applications have around *6x* reduction in the shuffle data. These applications do not require as much network throughput as the *IO-intensive* applications. The *CPU-intensive* application has around five orders of magnitude reduction in the shuffle data. Therefore, this application requires even less network throughput.

For experiments, we use Pivotal's AWB, which is a 1000-node cluster consisting of a 540-node Hadoop (version 2.2.0) installation [4]. Each node consists of 2 six-core Intel Westmere CPUs with a total of 24 threads, 48GB memory, 2 disks each with 2TB capacity, and 10 disks. The nodes use infiniband for network connectivity. This setup has a total memory capacity of 12 TB and HDFS storage capacity of 10 PB. There are 3 dataset sizes used in this work. They are named small, medium, and large in the order of increasing data size. The datasets for terasort were created using the *teragen* application. Datasets for remainder of the applications were generated by copying the 300GB datasets of PUMA multiple times. The smallest dataset size is 16.7TB and the largest is 61.5TB.

IV. Data size scaling

The original design of MapReduce is based on the premise that the framework allows application to scale over a large commodity cluster [2]. As the types of applications that run in MapReduce have become more diverse [3], characterizing data scaling helps to identify application specific properties that may or may not affect its scaling. Data size scaling determines how applications scale when data size changes but the cluster resources do not. It is the most applicable form of scaling compared to other scaling such as scale out or scale up because once a cluster is designed its hardware is not upgraded frequently.

Strong and weak scaling are alternative approaches to analyze the scalability of an application. In strong scaling, total data size is kept the same while the number of nodes is increased. In weak scaling, data size assigned to each node is kept constant while the number of nodes is increased. A strong scaling application is the one whose ideal runtime decreases proportional to the number of nodes, whereas ideal runtime is constant in a weak scaling application. In our study, we use a fixed map and reduce task data size similar to weak scaling. In addition, we also perform other Hadoop specific changes to ensure that the application's performance is not by affected resource contention overhead among the map and reduce tasks. Our data scaling methodologies are as follows.

Applications	Small	Medium	Large
terasort	2400	6000	9000
rankedinvertedindex	2400	6000	9000
word count	600	1200	1800
invertedindex	600	1200	1800
termvectorperhost	600	1200	1800
grep	4	7	10

Table III: Number of reduce tasks for the three data sizes.

1) Per map task computation: The data scaling experiments use same default split size of 128MB for all data sizes. Split size is the size of a block of data in HDFS. As as map task processes one block of data, this ensures that the computation per map task for all datasets is same. This avoids any computation imbalance among the map tasks.

2) Per reduce task computation: While the total number of map tasks is determined by the number of splits in the input data, the number of reduce tasks is a user configuration variable. Therefore, to avoid any computation imbalance among the reduce tasks the total number of reduce tasks is proportionately increased for all the datasets. This keeps the data size processed by a reduce task same across all input data sizes for an application. Table III shows the number of reduce tasks used for all applications for all datasets. The ratio between the number of reduce tasks for the datasets is similar to the ratio between their sizes. For example, the dataset size ratio for *rankedinvertedindex* for medium to small dataset is 15.90TB/5.90TB=2.83, and the ratio between the number of reduce tasks is 2400/750=3.2.

3) Application master: When the application master is co-scheduled with map and reduce tasks in the same node, it can cause slowdown due to increased resource contention on the application master node. In order to minimize these effects, we increase AM's memory to occupy a single node. This avoids the resource contention problem. It also ensures that the application master has sufficient memory to keep track and manage the large number of map and reduce containers.

4) Parallelism and overlap configuration: Data scaling experiments use concurrent container slot (CCS) of 16 which runs a maximum of 16 concurrent map tasks or 4 concurrent reduce tasks. The CCS of 16 was used in order to avoid slowdown that may occur when running too many concurrent map tasks. The performance effects of changing CCS values is discussed in Section V. Additionally, slowstart value in these experiments is 50%. This value does not cause any adverse performance effects when used with CCS of 16.

A. Scaling results

Table IV shows that the execution time per unit data (e.g., $\frac{T_S}{D_S}$) for *IO-intensive* applications *terasort* and *rankedinvertedindex* increases as the dataset size increases. For *terasort*, the execution time per unit data for large is 126.7 seconds per terabyte (TB), which is *1.8x* that of 71.4 for small. The execution time per unit data steadily increases for *terasort* as dataset size increases from small to medium to large. *Terasort* does not scale well with the increase in data size. Computation per unit data increases as data size increases. *Rankedinvertedindex* has similar results for execution time per unit data for increasing dataset size.

In contrast, for *Balanced* and *CPU-intensive* applications (wordcount, invertedindex, termvectorperhost, and grep) Table IV shows that computation per unit data remains closely similar as dataset size increases. The computation time is within *1.2x* for the small and the large datasets for all the four applications. For wordcount and invertedindex, the cost per unit data increases by *1.2x*, for termvectorperhost it decreases by *1.2x* and for *grep* it decreases by *1.04x*. This implies that as the data size increases these applications scale well. Below, we perform resource usage analysis to understand why the three categories of applications show different scaling behaviors.

B. Resouce usage analysis

Table V shows the average values of CPU, disk IO, and network usage of a node in the cluster. The CPU utilization, shown as CPU in the table, is the average user CPU value. CPU utilization of a system is divided into user, system, iowait, and idle. The table shows the user CPU usage because it represents time spent on CPU by user programs and the main user program running in the AWB cluster is Hadoop. There are two types of disk IO shown in the table. IO in the table represents the average of bytes read and written to disks throughout the entire application run. IO_M in the table represents the average only during the map phase execution. The main reason to include both disk IO values is to distinguish between the effect of reduce phase on the overall throughput. Most IO operations are performed on map phase. Due to this reason, as reduce phase becomes longer for larger datasets, the average IO throughput decreases when averaging over the entire duration of an application's execution. The network utilization, shown as Net in the table, is measured by taking the average of network input and output bytes during the reduce phase of an application. We avoid including map phase time in the network average calculation because of the non existent network usage during map phase. Paragraphs below explain the results for each of the resource values.

CPU usage. CPU usage of all six applications are fairly similar for all the datasets. For example, CPU usage values of *grep* are the highest at 65%, 65%, and 64%. With same value for CCS and SS, the number of containers running at any moment during the application's execution is similar for all datasets. This results in similar CPU usage values. CPU usage values in map phases are relatively higher than the averages shown in the table. This is because reduce phase is less CPU intensive than map phase.

IO usage. The average disk utilization during map phase (IO_M) of all six applications are also fairly similar for all

Applications	Small			Medium			Large		
	D_S **(TB)**	T_S **(s)**	$\frac{T_S}{D_S}$	D_M **(TB)**	T_M **(s)**	$\frac{T_M}{D_M}$	D_L **(TB)**	T_L **(s)**	$\frac{T_L}{D_L}$
terasort	17.60	1257	**71.4**	43.90	4479	**102.0**	61.50	7791	**126.7**
rankedinvertedindex	16.70	1115	**66.8**	38.60	3385	**87.7**	55.30	6068	**109.7**
wordcount	19.40	761	**39.2**	38.30	1790	**45.8**	57.70	2778	**47.6**
invertedindex	19.40	756	**39.0**	38.30	1893	**48.6**	57.70	2775	**48.1**
termvectorperhost	19.40	1462	**75.4**	38.30	2550	**66.6**	57.70	3566	**61.8**
grep	19.40	702	**36.2**	38.30	1348	**35.2**	57.70	1988	**34.5**

Table IV: Completion times (T_S, T_M, T_L) and computation time per unit data for data sizes.

Applications	CPU (%)			IO_M (MB/s)			IO (MB/s)			Net (MB/s)		
	S	M	L	S	M	L	S	M	L	S	M	L
terasort	22	21	18	219	219	215	188	130	112	89	63	45
rankedinvertedindex	28	24	25	211	200	207	172	123	107	94	64	46
wordcount	50	40	42	99	76	87	85	76	81	31	37	27
invertedindex	61	54	48	95	85	72	91	77	67	41	32	34
termvectorperhost	47	44	42	82	72	65	43	46	50	15	17	29
grep	65	65	64	67	70	68	63	66	63	5	6	6

Table V: Average resource utilization of a single node in AWB (Datasets: S=small, M=medium, L=large).

datasets. The values for the *IO-intensive* applications are particularly significant as both their input and output data size is relatively high. *Terasort* IO_M is 219 MB/s, 219 MB/s, and 215 MB/s for small, medium, and large datasets respectively. *Rankedinvertedindex* shows similar values. This suggests that as the dataset size increases, the disk IO throughput is not affected. This is because the IO operations are mostly performed in map phase, and with the same CCS and SS values, there is no additional IO load during map phase for any of the datasets. Another disk throughput value (IO), which is the average for both map and reduce, is lower than the corresponding IO_M. This is more noticeable for larger datasets because reduce tasks do not perform much IO operations and the longer reduce time for larger datasets lowers the average value.

Network usage. Out of all applications, for the two *IO-intensive* applications – *terasort* and *rankedinvertedindex*, the network throughput decreases as the dataset size increases. For *terasort*, the network usage for small is double that of large. Similarly, for *rankedinvertedindex*, the network usage halves from 94 MB/s to 46 MB/s for large dataset compared to the small one. For the remaining four *Balanced* and *CPU-intensive* applications, network throughput remains similar for all dataset sizes and are not as different as for *IO-intensive* applications.

Figure 1 shows the network usage of *terasort* throughout its execution for the three datasets. It also shows that the even though the number of concurrent reduce tasks and reduce input size is same, the network usage peaks at 140MB/s, 100MB/s, and 75MB/s respectively for the three datasets.

In summary, the data scaling results showed that *Balanced* and *CPU-intensive* applications scale proportionally to the increase in dataset size. But, *IO-intensive* applications do not. The resource usage analysis showed significant decrease in network throughput for *IO-intensive* applications, whereas other applications did not show such decrease. CPU utilization and IO throughput for all applications were fairly similar. This suggests that *IO-intensive* applications suffer slowdown with increase in dataset size due to the decrease in overall network throughput.

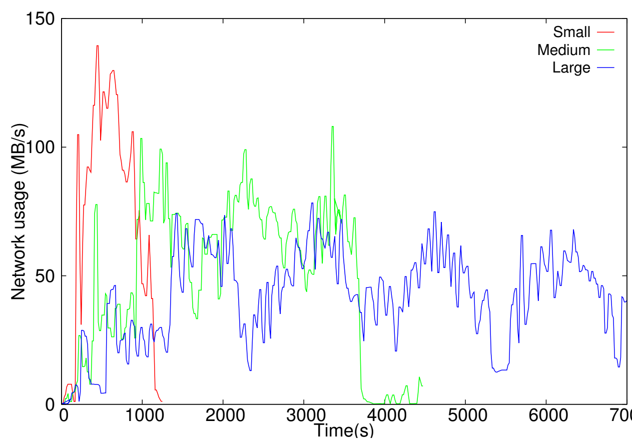
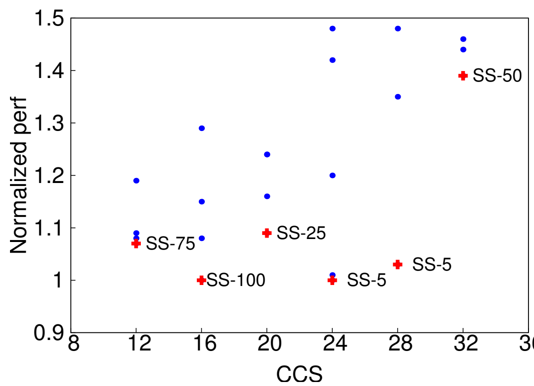


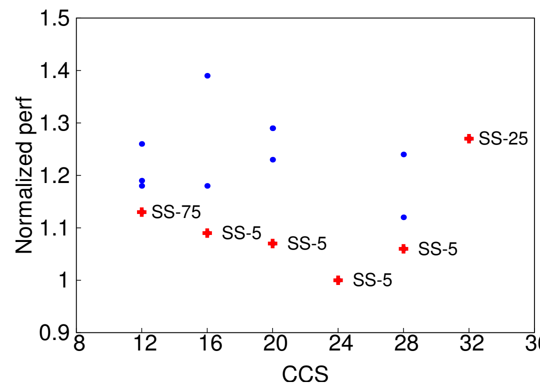
Figure 1: Network usage of *IO-intensive* application *terasort*.

V. Parallelism and overlap

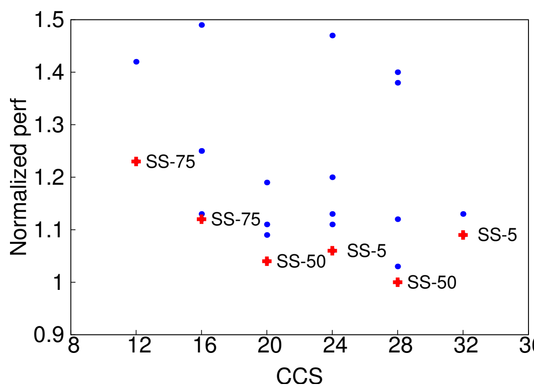
Our previous study on a 10-node cluster shows that *IO-intensive* application performanace is better with lower map task parallelism, *Balanced* applications with higher, and *CPU-intensive* with medium map task parallelism [8]. In this work, we repeat this systematic study of performance of applications for different CCS values on a large cluster. Additionally, we also study the effect of map and reduce overlap by changing the values of slowstart (SS). As mentioned before in Section II-B, a high CCS may result in CPU or IO bottleneck due to too much parallelism, whereas a low CCS results in too little parallelism resulting in underutilization of CPU and IO resources. Similarly, a high



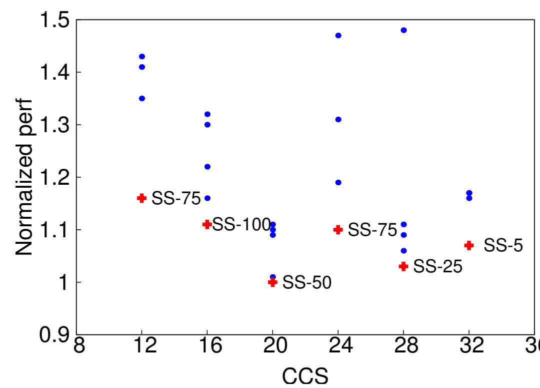
(a) Terasort.



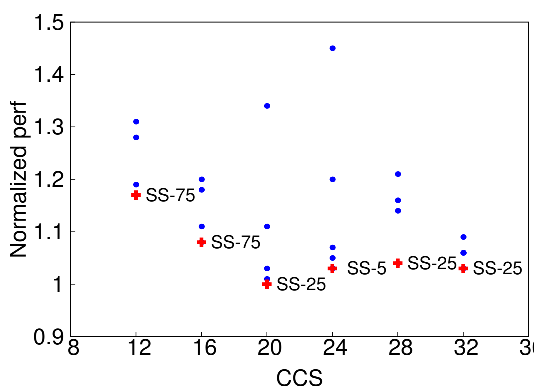
(b) Rankedinvertedindex.



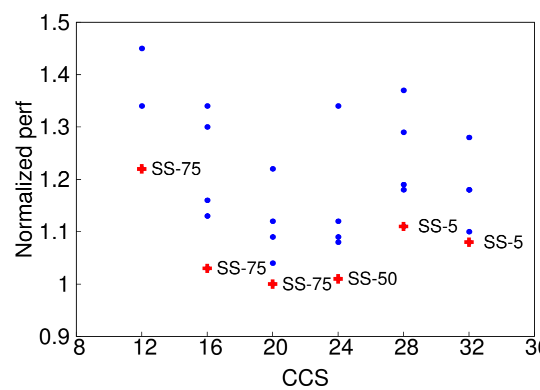
(c) Wordcount.



(d) Invertedindex.



(e) Termvectorperhost.



(f) Grep.

Figure 2: Best performing combinations of CCS and SS for all six applications.

SS value, which has less overlap between map and reduce tasks, can result in excess time spent copying data after map tasks have completed as well as increase in overhead to store intermediate data. In contrast, a low SS value can start reduce tasks too early leading to starvation of the reduce tasks. As reduce tasks do not process the data until output of all map tasks are available, they occupy containers that decreases the effective parallelism for map computation and lowers utilization.

Figure 2 shows the normalized performance values for CCS and SS combinations for all six applications for the small dataset. The normalized performance value for a CCS and SS combination is the relative completion time when compared to the fastest time. For the experiments, CCS values range from 12 to 32 in increments of 4. CCS values below 12 and above 32 are not included in this paper because they never were the best case. SS values used in the experiments are 5, 25, 50, 75, and 100. The default value of SS in Hadoop is 5 and the maximum value for SS is 100. The CCS value is shown on the x-axis. Each vertically aligned point represents a test at that CCS with a different SS value. The best performing SS value for each CCS is labelled in the figure. For readability, other points are not labelled. The y-axis cuts off at 50% slowdown. Points > 1.5 are not plotted. The performance effects of these parameters are explained below.

IO-intensive. Figures 2a and 2b show the best CCS and SS values for *IO-intensive* applications *terasort* and *rankedinvertedindex*. For *terasort*, CCS of 16 and 24 perform best with SS at 100 and 5, respectively. For *rankedinvertedindex*, the best CCS and SS values are 24 and 5. A SS of 5 lowers the map task parallelism because reduce tasks occupy containers that would otherwise go to maps. On the other hand, when SS is 100, there is no map and reduce overlap, therefore map tasks occupy all the containers. *IO-intensive* applications perform better with lower map task parallelism [8]. Therefore, these applications have best performance in two cases – low SS with high CCS and high SS with low CCS. The figure shows this behavior.

Balanced. Figures 2c, 2d, and, 2e show the best CCS and SS values for *Balanced* applications *wordcount*, *invertedindex*, and *termvectorperhost*. For *wordcount*, the best CCS and SS values are 28 and 50. For *invertedindex*, they are 20 and 50. For *termvectorperhost*, they are 20 and 25. These values show the best CCS is either 20 or 28 and best SS is either 25 or 50. This suggests that *Balanced* region applications have best CCS values close to the number of CPU threads and SS values in the medium range.

CPU-intensive. Figures 2f shows the best CCS and SS values for *CPU-intensive* application *grep*. It has the best CCS and SS of 20 and 75. The best CCS of *CPU-intensive* region is comparable to *Balanced* region but the best SS value is higher at 75. As the reduce input of *grep* is very small, there is no benefit with early shuffle. Therefore the best SS is higher. *CPU-intensive* applications perform better with map parallelism close to the number of CPU threads [8]. Therefore, with CCS 20, map parallelism is close to the number of CPU threads of 24. One interesting observation is for CCS of 28 and 32 where the best SS is 5. This happens because, as discussed earlier, starting reduce tasks early decreases map task parallelism. With SS 5, the effective map task parllelism for CCS of 28 and 32 is closer to that at 20.

In summary, all six values of CCS and all five SS used in the experiments were best for at least one application. This indicates that applications have different suitable levels of parallelism and overlap. Therefore, it is necessary to know how to select these values to obtain best performance. Our findings show that, *IO-intensive* applications *terasort* and *rankedinvertedindex* have best performance for lower CCS, higher SS and higher CCS, lower SS. *Balanced* applications *wordcount*, *invertedindex*, and *termvectorperhost* have best performance for medium CCS and SS values. *CPU-intensive* application *grep* has best performance for medium CCS value and higher SS value.

Applications	#maps	#reduces	2GB	3GB	4GB	5GB	6GB	7GB
terasort	359559	6000	M	M	O	P	P	✓
rankedinvertedindex	335355	6000	M	M	O	P	P	✓
wordcount	317661	1200	M	O	P	✓	✓	✓
invertedindex	317661	1200	M	O	P	✓	✓	✓
termvectorperhost	317661	1200	M	O	P	P	✓	✓
grep	317661	7	M	M	O	✓	✓	✓

M: failure during map
O: failure during map and reduce overlap
P: partial failure
✓: Successful completion

Table VI: Different memory sized AM for medium dataset.

Applications	INFO		WARN		ERROR	
	Count	%	Count	%	Count	%
terasort	26416168011	99.98	3235387	0.01	32059	0.00012
rankedinvertedindex	24921335221	99.98	2995487	0.01	31203	0.00013
wordcount	25842415754	99.99	2968107	0.01	708	0.000003
invertedindex	25879040993	99.99	2966992	0.01	31409	0.00012
termvectorperhost	27540728526	99.99	2971957	0.01	807	0.000003
grep	27563866911	99.99	2979514	0.01	383	0.000001

Table VII: Number and percentage of INFO, WARN, and ERROR log messages for large dataset.

VI. Experiences on the large cluster

Our study of AWB helped to shed light on issues that are not apparent in smaller clusters. This section describes two experiences gained from the study of AWB. These experiences show issues related to application performance in a large cluster.

A. Application master sizing

As described in Section II-A, an AM negotiates for containers with *resourcemanager* and coordinates the execution of map and reduce tasks in the allocated containers. When processing large datasets, an AM may run out of memory due to the tracking needs for the large number of map and reduce tasks. Table VI shows the completion status for medium dataset by AM size. The CCS and SS settings are 16 and 50. The default memory size of AM is 2GB. For 2GB, AMs of all applications fail during map phase. They run out of memory. For 3GB also all applications fail. However, three of them fail due to out of memory exceptions during overlap of map and reduce phase. If an AM fails, YARN will run it one more time. We denote a run that succeeds on an AM restart as a partial failure. For 4GB, three applications partially fail. For the three applications, the second time AM is run, the applications complete successfully. [1] All applications run successfully consistently only for 7GB AM size. Our study shows that, appropriately sizing AM is crucial to successful completion of an application.

[1]The partial failure is an empirically observed result and was not consistently repeatable. A "P" in the table indicates at least one time a partial failure was observed. It does not imply the result is always a partial failure.

B. Log aggregation

Map tasks, reduce tasks, and application master generate logs during execution. YARN aggregates these logs by copying them from *nodemanager*s to HDFS. There are eight log levels in Hadoop, which in order of increased verbosity are OFF, FATAL, ERROR, WARN, INFO, DEBUG, TRACE and ALL. The default verbosity level is INFO. For a large cluster such as AWB, the log level of INFO produces large amount of log information. The log size for large dataset is 5TB and small is 50GB. Having large amount of log helps to troubleshoot an application. On the other hand, it creates an overhead of storage and HDFS traffic that may interfere with the application reading and writing operations.

Table VII shows the breakdown of the log types when using INFO log level for large dataset. With INFO log level, an application produces logs with INFO, WARN, ERROR, and FATAL levels. Each application generates around 26 billion INFO log lines, which is around 99.99% of the total log data. Next smaller log type is WARN, which is around 0.01%. The number of ERROR logs are very less, around one thousandth of a percentage. This shows that INFO log level is highly verbose, whereas as the next log level WARN is highly sparse. This suggests that an intermediate log level that provides relevant troubleshooting information may be necessary to reduce the log data size.

VII. Conclusion

In this paper, we evaluated three different aspects of running a MapReduce application on the 540-node Pivotal's AWB Hadoop cluster. The first evaluation shows that the CPU and IO characteristics of an application determines in whether an application may or may not scale when the datasize increases. We found that *Balanced* and *CPU-intensive* applications scale proportionally to the increase in data size, whereas *IO-intensive* applications do not. The resource usage analysis shows that for *IO-intensive* applications the slowdown is caused by decrease in network throughput. The second evaluation shows applications requiring different amounts of parallelism (CCS) and overlap (SS) to minimize completion time. The third evaluation shows two observations - the application master memory size is determined by the dataset size and the default log level of INFO produces large log size and is not suitable for a large cluster. Our findings in this paper thus have helped to understand data scaling behavior, performance effects of parallelism and overlap, and expected issues when running MapReduce applications in a large cluster.

Acknowledgements

Portions of the research in this paper use results obtained from the Pivotal Analytics Workbench, made available by Pivotal Software, Inc.

References

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

[2] J. Dean and S. Ghemawat, "MapReduce : Simplified Data Processing on Large Clusters," in *Proc. of OSDI*, 2004.

[3] "Large scale Hadoop clusters," https://wiki.apache.org/hadoop/PoweredBy.

[4] "Analytics workbench," http://www.analyticsworkbench.com/.

[5] C. Reiss, A. Tumanov, G. R. Ganger, R. H. Katz, and M. A. Kozuch, "Heterogeneity and dynamicity of clouds at scale," in *Proc. of SOCC*, 2012.

[6] Y. Chen, A. Ganapathi, R. Griffith, and R. Katz, "The Case for Evaluating MapReduce Performance Using Workload Suites," in *MASCOTS*. IEEE, 2011.

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

[9] "HDFS architecture," http://hadoop.apache.org/docs/r2.2.0/hadoop-project-dist/hadoop-hdfs/HdfsDesign.html.

[10] 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 *Proc. of SOCC*, 2013.

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

[12] A. Kawa, "Hadoop Adventures at Spotify ," *Hadoop World*, 2013.

[13] M. Sevilla, I. Nassi, K. Ioannidou, S. Brandt, and C. Maltzahn, "A framework for an in-depth comparison of scale-up and scale-out," in *Proc. of DISCS*, 2013.

[14] R. Appuswamy, C. Gkantsidis, D. Narayanan, O. Hodson, and A. Rowstron, "Scale-up vs Scale-out for Hadoop: Time to Rethink?" in *Proc. of SOCC*, 2013.

[15] R. McCreadie, C. Macdonald, and I. Ounis, "MapReduce indexing strategies: Studying scalability and efficiency," *Information Processing & Management*, 2012.

[16] D. Jiang, B. C. Ooi, L. Shi, and S. Wu, "The performance of MapReduce: an in-depth study," *Proceedings of the VLDB Endowment*, 2010.

[17] K. V. Shvachko, "Apache Hadoop: The Scalability Update," in *USENIX;login*, 2011.

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

[19] S. Huang, J. Huang, J. Dai, T. Xie, and B. Huang, "The HiBench benchmark suite: Characterization of the MapReduce-based data analysis," in *ICDEW*, 2010.

[20] B. F. Cooper, A. Silberstein, E. Tam, R. Ramakrishnan, and R. Sears, "Benchmarking cloud serving systems with YCSB," in *Proc. of SOCC*, 2010.

[21] D. Menasce, "TPC-W: a benchmark for e-commerce," *IEEE Internet Computing*, 2002.