DEIM Forum 2015 E7-3

An Improvement on Hadoop Scheduling by Utilising Analysed CPU Resource Demands

Kun LIU†, Daisaku YOKOYAMA††, Masashi TOYODA††, and Masaru KITSUREGAWA†††,††

† Graduate School of Information Science and Technology, the University of Tokyo
Hongo, Bunkyo-ku, Tokyo 113-8656, Japan
†† Institute of Industrial Science, the University of Tokyo
4-6-1 Komaba, Meguro, Tokyo 153-8505, Japan
††† National Institute of Informatics
2-1-2 Hitotsubashi, Chiyoda-ku, Tokyo 101-8430, Japan
E-mail: †{liukun_oliver,yokoyama,toyoda,kitsure}@tkl.iis.u-tokyo.ac.jp

Abstract In Hadoop, tasks of I/O-intensive workloads require much less CPU resources than CPU-intensive tasks, yet current CPU scheduling assumes that each task equally saturates one physical core. Such policy could lead to under-utilisation of CPU resources. We alleviate this problem by proposing the Finer Grained CPU Scheduler that takes into account the actual CPU resource demands of tasks. Demand information is acquired by analysing sample tasks, and then utilised in the scheduling. To evaluate our approach, we design a practical scheduling system, implement it in existing Hadoop framework, and conduct a series of experiments running mixed workloads. The results demonstrate that compared to the state-of-the-art YARN approach, our method improves the throughput of CPU-intensive workloads by approximately 20%, and does not compromise the performance of I/O intensive jobs.

Key words Hadoop, CPU scheduling, resource demands

1. Introduction

Hadoop MapReduce has become the de facto standard for large scale data analytics. Since the advent of Hadoop, constant effort has been made to improve its efficiency of cluster resource utilisation. A notable breakthrough is YARN [1] (Yet Another Resource Negotiator, the next generation of Hadoop compute platform developed by Yahoo!), which revolutionised the way of memory allocation: instead of statically limiting the number of parallel MAP/REDUCE tasks per node, it sets the per-node memory capacity and schedules each task a configurable amount, i.e., YARN schedules "memory resources" rather than "static task slots". However, different from this relatively flexible and efficient memory management, current CPU scheduling is very coarse.

In Hadoop YARN, the **Resource** class models memory and CPU virtual cores (*vcore*). For each datanode the vcore capacity is configured equal to its number of physical cores [2]; each task requests exactly one vcore. This policy is loosely akin to the static task slots in classic Hadoop, the difference being: while classic Hadoop views every task as completely equal, YARN views them equally CPU-wise. In practice, however, there are both CPU- and I/O- intensive workloads [5]. For a task spending most of the time doing

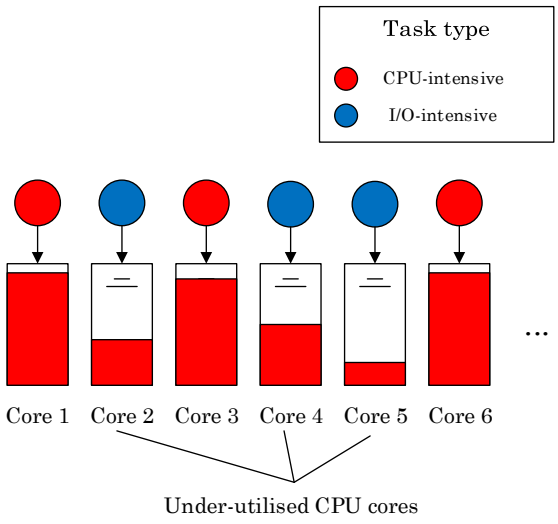

Figure 1: CPU under-utilisation in current Hadoop

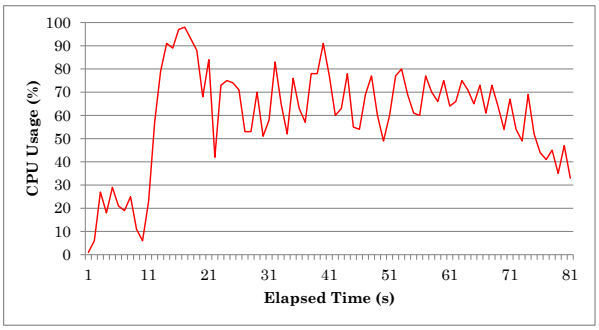

Figure 2: CPU load on the datanode

I/O operations, assuming it also saturates one physical core is by no means reasonable. As illustrated in Figure 1, for each physical core assigned to an I/O intensive task, a large proportion of the processing power is left unutilised, which is a huge waste of valuable CPU resources.

To verify such under-utilisation in practice, we performed a simple test on a datanode. The experimental environment is shown in Table 2. On the node which had 8 physical CPU cores, we executed a *TeraGen* job that contained 8 tasks. *TeraGen* is a highly I/O-intensive workload that generates random data for the Terabyte sort benchmark, therefore we expected a proportion of CPU resources to be wasted. As shown in Figure 2, the average CPU load of the datanode throughout the test was barely 60%, which implied that approximately half of the CPU processing power on the node was left unutilised.

To tackle such problem, we propose a Finer Grained CPU Scheduler (FGCS). The basic idea of our approach is to schedule CPU resources according to the actual needs of tasks. The primary contributions in this paper are:

（1） Designing a new CPU resource model that effectively improves CPU utilisation by considering the node capabilities and task demands

（2） Designing and implementing the features to automatically estimate the CPU resource demands of tasks, and utilise such information in task scheduling.

The rest of the paper is organised as follows. Section 2 describes the FGCS in details. Then we summarise the experiments to evaluate our approach in Section 3. Next, we introduce a few related researches in Section 4. Finally we conclude the paper in Section 5.

2. Approach

The problem of current CPU scheduling in Hadoop is the assumption that every task would saturate one physical core, despite the fact that many tasks, especially I/O intensive ones, require much less CPU resources. To enable a Finer Grained scheduling that could fully utilise yet not over-stress the CPU resources, we need to model the processing power demands of the tasks, or more specifically, the proportion of the processing power of one physical core that each task roughly consumes. In addition, the actual CPU resource demands are unknown before the tasks are executed. In our scheduling method, we parametrise the CPU resource demand of each task by its "CPU intensity", and adopt an analysing based approach to obtain the demand information.

For clarification, in this paper, a *workload* refers to the definition of a job, or more specifically, the definition of the Mapper and Reducer classes. In programming language terms, a workload could be viewed as a class, and a job could be viewed as a specific object of the class. Two jobs belong to the same workload iff they have the exact same Mapper and Reducer class definitions.

2.1 CPU resource model

In order to quantify the CPU resources as "proportions of one core", we define the capacity of each core as 100, representing 100% of its processing power. Given n, the number of physical cores of each node, the total CPU resource capacity per node is

$$Capacity = n \times 100 \tag{1}$$

With Equation 1, the default CPU resource request for a task could be straightforwardly set to 100, independent of the number of cores per node. This becomes especially convenient when: (a) the per-task CPU resource demand of a job is unknown, e.g., when the workload has not yet been analysed; (b) some tasks are so short that we do not bother to analyse them. In either case, we could request 100 CPU resource for each task, assuming it saturates one physical core as YARN currently does, in order to avoid too many parallel tasks over-stressing the CPU resources of some nodes.

For a particular task t, its CPU resource demand, i.e., the percentage of one core it roughly saturates, could be estimated by the "CPU intensity" of t. We define CPU intensity as follows due to its inverse relationship with I/O intensity:

$$d_t = \frac{CPU\ time}{CPU\ time + I/O\ time} \times 100 \tag{2}$$

the intuition is, the more CPU-intensive t is, the higher the value of d_t, e.g., if t is doing CPU computations only throughout its entire course, d_t is estimated as 100 meaning t saturates one physical CPU core.

In practice, tasks have a-priori unknown CPU resource demands. We therefore analyse each workload by executing a sample job that contains a small number of tasks, and calculating the average demand. This is practical since in Hadoop, the same workload tends to be repeatedly executed, according to Ren Kai et al. [10]. The overhead of the small sample job could thus be amortised over time.

For a particular task t, we could get its total CPU time in various ways, e.g., via "/proc/[pid]/stat" where pid is the process id of the task JVM. Getting the exact "I/O time" directly is difficult, but it is possible to keep track of the amount of I/O operations. We strive not to underestimate the CPU resource demands, since otherwise we could end up assigning more CPU workloads than we ought to, potentially over-stressing some nodes. Due to this consideration we calculate the I/O time as if t always had optimal I/O performance, i.e., the full disk bandwidth of the node task t is reading from or writing to. Since Hadoop usually runs on non-overlapping FileSystem(注1) and HDFS, the I/O time in Equation 2 becomes

$$I/O\ time = \frac{FileSystem\ bytes}{Disk\ I/O\ rate} + \frac{HDFS\ bytes}{HDFS\ I/O\ rate} \quad (3)$$

The disk and HDFS rate could be obtained by conducting a series of I/O tests.

Once the sample job has been executed, we could estimate the per MAP/REDUCE task CPU resource demand of the workload as

$$d_{MAP} = \overline{d_{MAP_i}} \quad (*)$$

$$d_{REDUCE} = \overline{d_{REDUCE_i}} \quad (**)$$

2.2 Demand analysing functionality

To make use of the CPU resource model defined in Section 2.1 we need to obtain

（1） The total CPU time of each task

（2） The total number of bytes read/written, including both FileSystem and HDFS, by each task

so that the CPU demand of each task could be calculated using Equation 2 and Equation 3. Then when a job finishes, its per MAP/REDUCE CPU demand could be calculated using Equation * and Equation **. When a submitted job is "known", i.e., the per MAP/REDUCE CPU demand of the workload has been calculated once, our system could take advantage of the demand information to enable the Finer Grained scheduling; otherwise the job could request the default per task CPU resource 100, as explained in Section 2.1.

In Hadoop, task scheduling is mostly automatic and transparent, e.g., with MapReduce the user only needs to define the job by the MAP and REDUCE functions, and rarely concerns about the details on the task level. This simplicity is a crucial advantage of Hadoop, as indicated by Lee et al. [8]. Moreover, the same task could have very different CPU resource demands in different environments due to diverse hardware performance, e.g., a CPU-intensive task could have lower CPU intensity if executed on datanodes with relatively more powerful CPU and slower disk. Consequently, workloads have to be re-analysed every time the cluster hardware changes. For simplicity and portability, we build the CPU demand analysis into Hadoop framework so that no extra effort (other than testing I/O rates when disk hardware changes, which is rather infrequent) is required by the Hadoop user. The rule of thumb is that the user does not need to write extra code for his MapReduce jobs.

(注1)：FileSystem refers to the local file system.

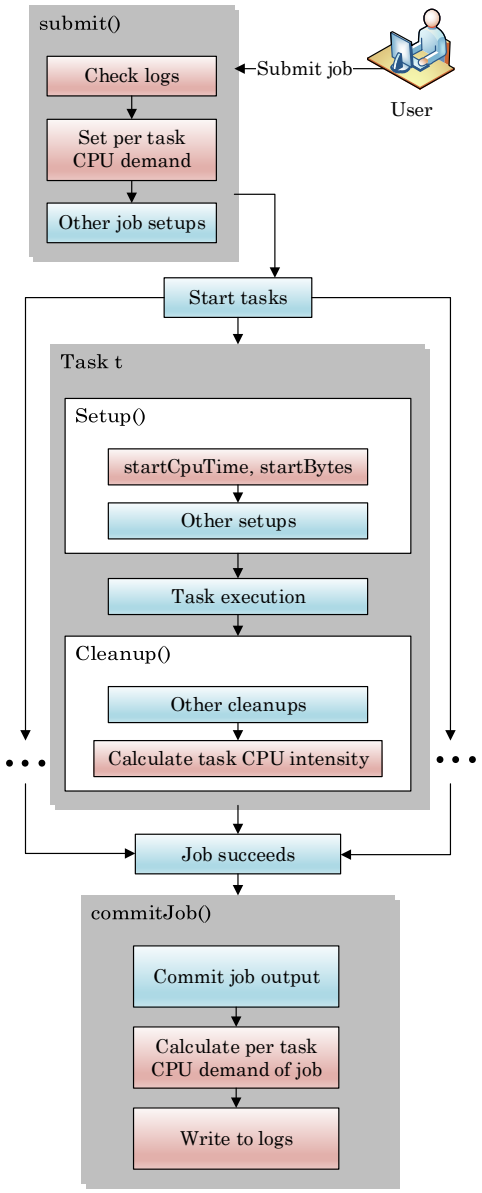

Figure 3: Simplified workflow of a MapReduce job with CPU demand analysing features built in

In the following subsections, we briefly review the workflow of a Hadoop MapReduce job. As we walk through each step, we present our objective and roughly explain relevant implementations. Figure 3 shows a simplified workflow with the analysing features built in. To distinguish from the Hadoop framework, we use a reddish colour for our implementations.

2.2.1 Job submission

When a job is submitted, FGCS wants to make use of the analysed resource demands if possible. Specifically, if

the job is "known", i.e., the same workload has already been analysed, it could set the per MAP/REDUCE CPU resource demand accordingly during the job initialisation; otherwise, it could request the default per-task CPU resource 100 as explained in Section 2.1.

In Hadoop, all job submissions need to go through the **submit()** method for job initialisations. At the beginning of **submit()**, FGCS checks the analysis logs in HDFS for previously analysed CPU resource demands of the same workload. In our system, two jobs are treated as the same workload iff they have the same name. Job name is also the identifier for searching CPU demand logs.

2.2.2 Task execution

For a particular task t, FGCS needs to obtain its total CPU time and total number of bytes read/written in order to calculate its CPU demand d_t. To do so FGCS gets the two values once before the task execution starts, one more time after the task finishes, and calculates the differences. Once the CPU demand has been calculated, the value needs to be stored in a robust and efficient way so that for each job, all the tasks that belong to it could be correctly put together to calculate the average.

In Hadoop, every Mapper/Reducer class inherits a base **Mapper/Reducer**. Our CPU demand calculations are implemented in these base classes, so that the features would be inherited to every MAP/REDUCE task. For clarity, we refer to the base classes as **MapperBase** and **ReducerBase** respectively for the remainder of this paper.

When t starts, the method **setup()** is invoked for task initialisations. At the beginning of **setup()**, t has virtually performed nothing. This is the point when FGCS fetches the CPU time of the task JVM and the number of bytes as the starting values. When t finishes, it calls the method **cleanup()** before exit. The very end of **cleanup()** is the point to fetch those two values again. Then we could calculate d_t using Equation 2 and Equation 3. In our system, I/O rates needed in Equation 3 are provided as properties in the Hadoop built-in configuration file "mapred-site.xml".

The value of d_t is kept in our self-defined task counters grouped in "CPUSTAT". Depending on the task type, we increment MAP_COUNT/REDUCE_COUNT by 1, and MAP_DEMAND/REDUCE_DEMAND by d_t.

Extra effort is made to make sure that the CPU intensity values are reliable

（1） If the task attempt fails we do not increment any counters, i.e., it is not involved in calculating the average per MAP/REDUCE CPU usage of the job it belongs to

（2） Ensuring that Combiners would not incorrectly increment the CPUSTAT counters. Combiner is a widely adopted optimisation which performs "local" reduce-type functions within the MAP tasks to decrease the amount of data passed to and eventually processed by REDUCE tasks. On one had, combiners are essentially Reducers: every **Combiner** class eventually inherits **ReducerBase**, and it is not unusual for a job to use its **Reducer** class as the Combiner. On the other hand, they are executed within MAP tasks. In other words, combiners are part of MAP tasks but they would increment REDUCE_COUNT and REDUCE_DEMAND if left unchecked. As such, we insert statements in **ReducerBase** so that those two counters are only incremented by real REDUCE tasks.

Workload	MAP	REDUCE
Pi	100	Default
Bbp	100	100
TeraGen	49	N/A
ETL	47	N/A

Table 1: Per MAP/REDUCE task CPU resource demand of each workload

2.2.3 Job completion

When all tasks finish, the per MAP/REDUCE CPU demand of the workload needs to be calculated and logged so that the information could be utilised for future scheduling. For reliability of the results, only jobs that succeeded are calculated.

In Hadoop, upon successful job completion the **commitJob()** method is called once for committing the output [9]. Within **commitJob()**, we should have the correctly aggregated CPUSTAT counters(注2) for the needed calculations.

At the end of **commitJob()** FGCS checks the analysis logs in HDFS. If a log with the same name already exists, indicating that the workload has already been analysed, the newly analysed results are discarded; otherwise, they are written into HDFS. If reanalysis for certain jobs are desired, the corresponding logs need to be explicitly cleared (using our script) so that new values could be written. This resembles many real-world MapReduce applications - the job does not overwrite existing paths/files.

2.3 Analysing workloads

With the functionalities described above, we analysed the CPU resource demands of the workloads used for experiments. Each workload was analysed with a 16-task sample job. Table 1 summarises the analysed CPU demands. For MAP-only workloads, the REDUCE column is blank.

When running sample jobs, we always set the per-task

(注2)：In Hadoop, task counters are maintained by each task. When the job finishes, for each counter the results are aggregated over all the tasks. Since **commitJob()** is only invoked if the job succeeds, we are guaranteed to have the correctly aggregated counters.

Hardware	
CPU	Xeon E5530, 2.40 GHz, 8 cores
Memory	24 GB
Disk	500 GB
Network	10 Gbps
Software	
Hadoop	2.6.0
Framework	YARN
Java	Oracle 1.7.0_21

Table 2: Experimental environment (per datanode)

CPU resource demand to the default value 100 since they were "unknown" at this point. In addition, some tasks are extremely short-lived, e.g., *Pi* contains a single REDUCE task that finishes in scant few seconds. For this kind of tasks, it is difficult to guarantee the accuracy of resource demand calculations as the overhead of task executing becomes non-negligible. Thus we ignored the analysed results and changed them to "default". As explained in Section 2.1 this default setting is a safe option to avoid overstressing the CPU resources.

3. Evaluation

To evaluate the FGCS, we conducted experiments on an 8-node cluster. The experimental environment is summarised in Table 2. Each of our datanode has 8 physical CPU cores, which according to current scheduling policy would the number of parallel tasks per node to 8, but by our standard would define the per-node CPU resource capacity as 8*100.

In practice, the Hadoop daemons on each datanode consume CPU resources, although the amount is usually insignificant. When HDFS is performing a lot of I/O operations, however, the relevant DataNode processes could have very high CPU usage. Due to this consideration, on each datanode we reserved one physical core for Hadoop daemons and the operating system, leaving a 7*100 CPU capacity for MapReduce tasks. This strategy resembled memory allocation in Hadoop: it is good practice to reserve a proportion of memory for the system on each node and leave the rest for actual MapReduce tasks.

3.1 Benchmarks

The per-task CPU resource demands of the benchmark workloads are summarised in Table 1. Except for *ETL*, all workloads came from the official benchmark of Hadoop [11]

（1） **Pi** The program estimated the value of π using the Quasi-Monte Carlo method

（2） **Bbp** This job computed x exact digits of π using the Bailey-Borwein-Plouffe formula. x was configurable, and the calculations were evenly distributed to all MAP tasks

（3） **TeraGen** In this job, each MAP task was specified to write 2 GB of random data to HDFS

（4） **ETL** We constructed this highly I/O intensive workload. Each MAP task read an HDFS block of plain text, converted it to XML, and wrote the result back to HDFS. This resembled many real-world Expand-Transform Load applications used for preprocessing in Big Data

Of these workloads, *Pi* and *Bbp* belonged to CPU-intensive category, while *TeraGen* and *ETL* were highly I/O intensive.

We conducted 3 groups of tests, which resembled 3 real-world workload patterns

（1） **Balanced Pattern** 1 CPU-intensive job ran along with 1 I/O-intensive job. There were 4 combinations of jobs

（2） **CPU-heavy Pattern** Each test contained 3 jobs: *Pi*, *Bbp*, and one of the I/O-intensive workloads. There were 2 different tests

（3） **I/O-heavy Pattern** Each test contained 3 jobs: *TeraGen*, *ETL*, and one of the CPU-intensive workloads. There were 2 different tests

The detailed job settings are summarised in Table 3. We tuned the number of tasks so that in each test, jobs would not have too different execution times (it would be unfair to assess a very large job and a tiny one, since the latter would have little impact on the overall performance).

	Balanced	CPU-heavy	IO-heavy
# CPU-intensive job(s)	1	2	1
# tasks / CPU-intensive job	70	50	50
# IO-intensive job(s)	1	1	2
# tasks / IO-intensive job	31	20	20

Table 3: Job settings for 3 groups of tests

3.2 Metrics

Firstly, for each CPU-intensive job, we assessed its average MAP time, i.e., the average execution time amongst all its MAP tasks(注3). CPU-intensive workloads heavily rely on the CPU resources, therefore if the cluster CPU resources were over-stressed, the performance of CPU-intensive tasks would be compromised.

Secondly, for every job, we assessed its job execution time, we expected that

（1） For each CPU-intensive job, the execution time would be reduced, since FGCS in general enables more parallel CPU-intensive tasks without compromising the individual task performance (one of our principles is not to over-stress the CPU resources)

（2） For each I/O-intensive job, the execution time would not change significantly. In our environment, even a sin-

(注3)：All our workloads had very small REDUCE tasks, or none, therefore it was meaningless to assess the REDUCE tasks.

Test	Workload	YARN	FGCS	Change (%)
$Pi + TeraGen$	Pi	78,186	78,066	0.15↓
$Pi + ETL$	Pi	78,563	78,039	0.67↓
$Bbp + TeraGen$	Bbp	90,937	90,981	0.05↑
$Bbp + ETL$	Bbp	90,489	90,875	0.43↑

Table 4: AvgMapTime (in ms, lower is better) of each CPU-intensive job under balanced pattern

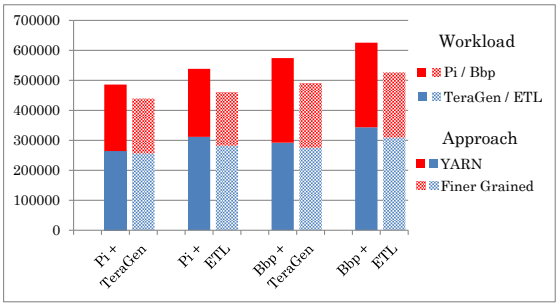

Figure 4: Job execution time (ms, lower is better) under balanced pattern

Test	Workload	YARN	FGCS	Change (%)
$Pi + TeraGen + ETL$	Pi	95,941	95,761	0.19↓
$Bbp + TeraGen + ETL$	Bbp	86,114	87,932	2.11↑

Table 5: AvgMapTime (in ms, lower is better) of each CPU-intensive job under I/O-heavy pattern

gle I/O-intensive task would nearly saturate the disk bandwidth on the relevant datanode, therefore more parallel I/O-intensive tasks would be unlikely to significantly improve the disk resource utilisation

3.3 Results

We chose Fair Scheduler as the memory scheduling policy so that in each test, both jobs could progress simultaneously. For each combination of workloads, we first executed the jobs under current YARN CPU policy, then switched to our Finer Grained approach.

3.3.1 Results under Balanced Pattern

Firstly, as shown in Table 4, compared to YARN, FGCS did not degrade the task performance of the CPU-intensive workload in any test. The differences between two approaches were tiny, well under 1%, thus negligible. This implied that FGCS did not over-stress the CPU resources.

Figure 4 shows the job execution times. For I/O-intensive workloads, there were no significant changes. However, the execution time of every CPU-intensive job was significantly reduced, by 18% up to 23%. This improvement was understandable, since FGCS allowed more CPU-intensive tasks with unaffected individual task performance.

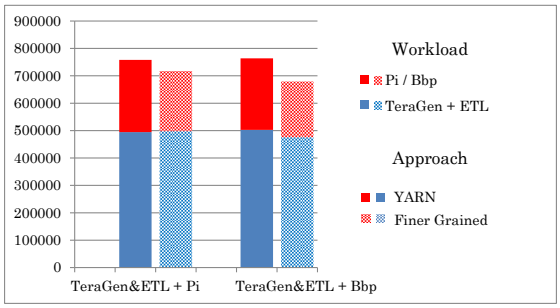

Figure 5: Job execution time (ms, lower is better) under I/O-heavy pattern

Test	Workload	YARN	FGCS	Change (%)
$Pi + Bbp + TeraGen$	Pi	78294	78224	0.09↓
	Bbp	87,525	87,585	0.07↑
$Pi + Bbp + ETL$	Pi	77,822	78,179	0.46↑
	Bbp	86,890	87,399	0.59↑

Table 6: AvgMapTime (in ms, lower is better) of each CPU-intensive job under CPU-heavy pattern

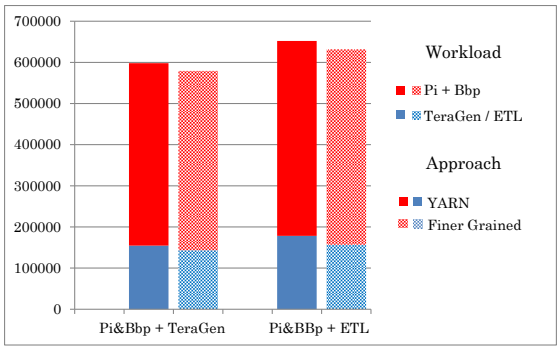

Figure 6: Job execution time (ms, lower is better) under CPU-heavy pattern

3.3.2 Results under I/O-heavy Pattern

As shown in Table 5, Bbp suffered a slight degradation in task performance, but we observed a slight performance improvement in $TeraGen$ and ETL, which well compensated the degradation. Therefore, the overall task performance did not drop in the $Bbp + TeraGen + ETL$ test.

The results of job execution time were rather consistent with the balanced workload pattern: as shown in Figure 5, there were no big changes in the I/O-intensive jobs, and the performance of the CPU-intensive job was improved by 17% up to 22%.

3.3.3 Results under CPU-heavy Pattern

Firstly, there were no performance degradation in individual tasks, as shown in Table 6.

Figure 6 shows the job execution times. Different from the balanced and I/O-heavy pattern, where CPU-intensive jobs were largely improved, in CPU-heavy pattern, FGCS

had almost identical performance compared to YARN. The reason was that under such pattern, most of the tasks were CPU-intensive, i.e., the proportion of wasted CPU resources were not significant. As a result, there was not much room for improvement on the CPU utilisation.

4. Related work

Others have realised the need to take into consideration the node capabilities such as processing power in scheduling. From Hadoop version 2.3.0 where vcore was implemented for the first time, it has been documented in Hadoop API docs that "a complementary axis for CPU requests that represents processing power will likely be added in the future to enable finer-grained resource configuration".

Motivated by the emergence of clusters with heterogeneous hardware, Gupta, Shekhar, et al. [4] proposed the ThroughputScheduler, which adopts a learning based approach to analyse the CPU and disk requirements of jobs, then schedules tasks based on the optimal matching of node capabilities and task needs, i.e., it attempts to assign CPU-intensive workloads to nodes with relatively more powerful CPU and disk-intensive workloads elsewhere. It effectively improves task performance in heterogeneous environment where task placements have significant impact.

Similarly, the CASH (Context Aware Scheduler for Hadoop), proposed by Kumar et al. [3], addresses the scheduling problem in a heterogeneous environment in three steps. Firstly, it classifies jobs into CPU-bound and I/O-bound category(注4). Then it classifies the nodes into "CPU buckets" and "I/O buckets", i.e., nodes with relatively more powerful CPUs/disks. Finally, it attempts to match CPU-/IO- intensive tasks to CPU/IO buckets.

While those approaches do consider the resource requirements of tasks, their basic logic "CPU-intensive tasks to faster CPUs, I/O-intensive tasks to faster disks" does not apply in a homogeneous cluster, which is still the common case. We consider the task requirements from a different point of view, i.e., the CPU resource requirements as opposed to the capacity. By scheduling CPU based on such requirements, our FGCS effectively improves the CPU utilisation - it alleviates CPU waste and avoids over-stressing the CPU resources.

Different from those approaches, FGCS does not change the task placement strategy. Our consideration is that current strategy is already complicated with many factors considered such as data locality [6] (assigning MAP tasks to nodes where the data to be processed is stored) . Data locality is so crucial that Facebook decided to adopt a Delay Scheduling [7] algorithm: if a job that should be scheduled next according to resource fairness, but it cannot launch a local task, it waits for a small amount of time, letting other jobs launch tasks first. Our approach does not conflict with those factors and works well with existing schedulers such as Fair Scheduler.

(注4) : More specifically, for each job, its MAP tasks and REDUCE tasks are separately classified, e.g., a job could be CPU-intensive in its MAP phase and I/O-intensive in its REDUCE phase.

5. Conclusion

Current CPU scheduling in Hadoop assumes that each task saturates one physical core, which is not the case for I/O intensive tasks. Consequently, a proportion of the cluster CPU resources are left unutilised. We proposed a Finer Grained CPU resource model that takes into account the actual CPU resource demands of tasks so that the wasted proportion could be effectively utilised. In particular, more CPU-intensive tasks could be executed in parallel without overstressing the CPU resources. As a result, Finer Grained approach significantly improves the throughput of CPU-intensive workloads and does not compromise the performance of I/O intensive ones.

To preserve the simplicity of Hadoop and for better portability, we designed and implemented the features into Hadoop framework to make use of the new CPU resource model. The only extra effort that our system requires compared to current Hadoop is testing the disk I/O rates once when the cluster hardware changes.

Future work could be extending the Finer Grained CPU policy to disk scheduling. Currently there are two challenges in implementing this: (1) the lack of a robust way to model the cluster disk resources; (2) the existence of HDFS making disk resources "non-local", i.e., a task on a node N could be reading from or writing to disks on other nodes. However, with major modifications on the resource model and deep delving into HDFS, emulating our approach to enable a Finer Grained disk scheduling is possible in the future.

References

[1] Vavilapalli, Vinod Kumar, Arun C. Murthy, Chris Douglas, Sharad Agarwal, Mahadev Konar, Robert Evans, Thomas Graves et al. "Apache hadoop yarn: Yet another resource negotiator," In Proceedings of the 4th annual Symposium on Cloud Computing, p. 5. ACM, 2013.

[2] Apache Foundation, Hadoop, `http://hadoop.apache.org`.

[3] Kumar, K. Arun, Vamshi Krishna Konishetty, Kaladhar Voruganti, and G. V. Rao. "CASH: context aware scheduler for Hadoop," In Proceedings of the International Conference on Advances in Computing, Communications and Informatics, pp. 52-61. ACM, 2012.

[4] Gupta, Shekhar, Christian Fritz, Bob Price, Roger Hoover,

Johan De Kleer, and Cees Witteveen. “ThroughputScheduler: Learning to Schedule on Heterogeneous Hadoop Clusters,” In ICAC, pp. 159-165. 2013.

[5] Hortonworks - typical Workloads Patterns For Hadoop, http://docs.hortonworks.com/HDPDocuments/HDP2/HDP-2.1.2/bk_cluster-planning-guide/content/typical-workloads.html

[6] Guo, Zhenhua, Geoffrey Fox, and Mo Zhou. “Investigation of data locality in MapReduce,” In Proceedings of the 2012 12th IEEE/ACM International Symposium on Cluster, Cloud and Grid Computing (ccgrid 2012), pp. 419-426. IEEE Computer Society, 2012.

[7] Zaharia, Matei, Dhruba Borthakur, Joydeep Sen Sarma, Khaled Elmeleegy, Scott Shenker, and Ion Stoica. “Delay scheduling: a simple technique for achieving locality and fairness in cluster scheduling,” In Proceedings of the 5th European conference on Computer systems, pp. 265-278. ACM, 2010.

[8] Lee, Kyong-Ha, Yoon-Joon Lee, Hyunsik Choi, Yon Dohn Chung, and Bongki Moon. “Parallel data processing with MapReduce: a survey,” AcM sIGMoD Record 40, no. 4 (2012): 11-20.

[9] Tom White. “Hadoop: The definitive guide,” O’Reilly Media, Inc., 2012.

[10] Ren, Kai, YongChul Kwon, Magdalena Balazinska, and Bill Howe. “Hadoop’s adolescence: an analysis of Hadoop usage in scientific workloads,” Proceedings of the VLDB Endowment 6, no. 10 (2013): 853-864.

[11] Hadoop benchmarks, https://github.com/apache/hadoop/tree/trunk/hadoop-mapreduce-project/hadoop-mapreduce-examples/src/main/java/org/apache/hadoop/examples.