This paper is author's private version of the paper published as follows:
Proc. 3rd Int'l Conf. on Networking and Computing (ICNC'12), 288-292

DOI: 10.1109/ICNC.2012.53

Dynamic processing slots scheduling for I/O intensive jobs of Hadoop MapReduce

Shiori KURAZUMI *, Tomoaki TSUMURA*, Shoichi SAITO* and Hiroshi MATSUO*

* Nagoya Institute of Technology
Gokiso, Showa, Nagoya, Aichi, 4668555, Japan
Email: shiori@matlab.nitech.ac.jp,
{tsumura,shoichi,matsuo}@nitech.ac.jp

***Abstract*—Hadoop, consists of Hadoop MapReduce and Hadoop Distributed File System (HDFS) , is a platform for large-scale data and processing. Distributed processing has become common as the number of data has been increasing rapidly worldwide and the scale of processes has become larger, so that Hadoop has attracted many cloud computing enterprises and technology enthusiasts. Hadoop users are expanding under this situation. Our studies are to develop the faster of executing jobs originated by Hadoop. In this paper, we propose dynamic processing slots scheduling for I/O intensive jobs of Hadoop MapReduce focusing on I/O wait during execution of jobs. Assigning more tasks to added free slots when CPU resources with the high rate of I/O wait have been detected on each active TaskTracker node leads to the improvement of CPU performance. We implemented our method on Hadoop 1.0.3, which results in an improvement of up to about 23% in the execution time.**

***Index Terms*—Hadoop; MapReduce; Scheduling algorithm; Slots Scheduling**

I. Introduction

Due to developing a variety of services on the Internet and explosively expansion of the population using the Internet, the number of data has been increasing rapidly worldwide and the scale of processes has become larger. We have taken the measure to improve overall performance. It is called distributed computing, which have cost-effective and general-purpose computers cooperate with each other to handle distributing data or processes all over the cluster consists of them. However, programming for distributed computing is highly complicated. There are several reasons. For example, we need to describe the way to deal with troubles during network communication and failures of components and how to partition, allocate, replicate data, and distribute processes over a number of computers. In order to describe simple algorithms without regarding of such programs for distributed computing, using Hadoop could provide us a good solution, with which we can skip to describe most part of the programs for distributed computing, such as communication recovery program, except for the original algorithm. Hadoop has attracted many cloud computing enterprises and technology enthusiasts. Both individuals and companies who use Hadoop have been increasing.

In general, Hadoop cluster consists of a large number of computers, so tasks assigned to TaskTrackers are not always Data-Local or Rack-Local. Furthermore, there is possibility that the physical distance between the node assigned a task to and another node maintaining the input data which the task requires may be longer in proportion to scale the cluster. In addition, increasing a communication delay for such reasons or sometimes other kind of failures may occur on Hadoop system, which makes task I/O wait time longer, hence CPU resources can not be used effectively. No matter how high I/O wait percentage is, default Hadoop scheduler can nothing to do except for waiting for the percentage to turn back naturally, because the slots set as a maximum number of tasks executed simultaneously are statically determined in Hadoop initializing procedure.

In this paper, we propose dynamic processing slots scheduling for I/O intensive jobs of Hadoop MapReduce to use CPU resources effectively by assigning more tasks to added free slots when it detects CPU resources with the high rate of I/O wait on each active TaskTracker node. We have implemented our method on Hadoop 1.0.3[1] and evaluating it.

The rest of this paper is organized as follows. A research background on Hadoop MapReduce is given in section 2. Our proposal and implementation is presented in section 3, and a performance evaluation in section 4. Finally, we indicate future works and conclude the paper in section 5.

II. Research Background

Hadoop is a platform for parallel and distributed large-scale data processing and one of the open-source projects by the Apache Software Foundation[2], which has been developed by the worldwide community of contributors. They enabled the framework to apply thousands nodes as components of Hadoop cluster and process petabyte scale data, therefore, achieved completing absolutely large-scale processing in a realistic executing time with their efforts. Hadoop consists of the two typical components: Hadoop Distributed File System (HDFS) mimicking Google File System (GFS) [3] and Hadoop MapReduce. The former is a distributed file system handling large-scale data and the latter is a framework for parallel and distributed processing [1], [4].

Hadoop MapReduce is an open-source implementation of MapReduce developed by Google[5]. Users submit jobs which

comprise of Map phase and Reduce phase. These are subsequently divided into each independent Map tasks or Reduce tasks. Input data which the job requires are also divided into units called input splits. Then the tasks are assigned to TaskTrackers on slave nodes consist in Hadoop cluster. The main advantage of Hadoop is to facilitate users to deal with large-scale parallel and distributed processing, thanks to more numerous divided tasks in proportion to scale the available nodes or CPU resources. MapReduce consists of the contents as follows: Single JobTracker on master node and multiple TaskTrackers on slave nodes. The former maintains and schedules tasks and latter actually process assigned tasks.

There are slots for task execution on MapReduce. Map slots are assigned Map tasks and Reduce slots are assigned Reduce tasks. These tasks have been selected by the job scheduler. The number of slots related to Map or Reduce, each can be determined on each TaskTracker. In other words, the number of slots is a maximum number of tasks executed simultaneously on each TaskTracker. Job schedulers have their roles in assigning appropriate tasks to such slots (Actually, they consider only that which TaskTracker requires tasks).

Original Hadoop provides the three job schedulers: Job Queue Task Scheduler, Fair Scheduler and Capacity Task Scheduler. Users can select which job scheduler among them. Job Queue Task Scheduler, which is the base of other job schedulers, is default job scheduler based on First In First Out （FIFO） queue. Tasks are assigned to nodes which maintain their input split with first priority (Data-Local) , or other nodes nearby such nodes which maintain their input split with second priority (Rack-Local) . Both of Fair Scheduler and Capacity Task Scheduler are job schedulers deal with multiple-users.

For example, due to Disk I/O or communication delay, I/O wait percentage of CPU may become higher, hence CPU resources can not be used effectively. As a consequence, obtaining required input splits and copying Map outputs take too much time, which makes the whole execution time of jobs longer. For these problems, default Hadoop assigns tasks based on the locations of their input splits by the job scheduler. Input splits are generally maintained all over the cluster. Job Scheduler called by JobTracker every HeartBeat communication on each TaskTracker selects tasks except for failed tasks or speculative tasks in that way described above. However, tasks assigned to TaskTrackers are not always Data-Local or Rack-Local. At this time, there is possibility that the physical distance between the node assigned a task to and another node maintaining the input data which the task requires may be longer in proportion to scale the cluster.

In order to avoid I/O wait caused by such factors, Shadi et al.[6] have proposed Maestro: a scheduling algorithm for Map tasks to be selected based on the number of hosted Map tasks and on the replication scheme for their input splits. Also, another scheduling algorithm considering whether allocate Non-Local tasks or not if the job scheduler can not select Data-Local tasks has been proposed by Xiaohong et al.[7]. Other related works to improve performance by scheduling tasks based on data locality have studied[8], [9]. These are the way to avoid I/O wait, therefore, different from our proposed method working after detecting I/O wait.

III. Dynamic processing slots scheduling for I/O intensive jobs

The goal of our studies is to reduce the whole execution time of Hadoop jobs by using each CPU resource effectively on slave nodes consist in Hadoop cluster. In this section, we introduce our concrete proposed method and details of the implementation.

A. Approach

At first, each active TaskTrackers on slave nodes consist in the cluster monitor the state of the CPU with reference to /proc/stat every predetermined interval. The varied states are shown in /proc/stat that how long time is spent by both of all CPUs and each CPU. We use I/O wait time information among them. Next, I/O wait percentage of each CPU is computed. While I/O intensive task is running on a TaskTracker, the I/O wait percentage of the CPU processing such task becomes higher. If this I/O wait percentage is greater than the predetermined threshold value, one Map slot is added (the number of Map slots is incremented) and which CPU has caused adding Map slot is recorded by TaskTracker. We have described what are Map slots in Sec.2. Then, if the state has been on over a certain period of time, which the I/O wait percentage of the CPU causing adding Map slot is smaller than the predetermined threshold value, one added Map slot is eliminated (the number of Map slots is decremented) . Note that the Map slots are up to the initial number of Map slots added the total number of CPUs of the slave node.

Job scheduler used for scheduling tasks to slots are called by JobTracker when HeartBeat communication occurs between a TaskTracker and a JobTracker. HeartBeat communication between each TaskTracker and a JobTracker occurs every 3 seconds. Hadoop implemented our method computes the I/O wait percentage and controls the number of Map slots based on the CPU information measured during the interval and the previous interval just before calling heartbeat method of JobTracker. The number of slots are maintained by each TaskTracker. The number of slots maintained by the TaskTracker causing calling the job scheduler is notified to the job scheduler when it is called. That is, the updated number of Map slots is notified every HeartBeat communication (Calling job scheduler) . The job scheduler finds out the appropriate Map tasks to the TaskTracker from remaining waiting Map tasks based on the updated number of Map slots and return them to JobTracker. If the number of Map slots has been increased, the job scheduler assigns more Map tasks to added free Map slots. As a result, the low usage CPU resources caused by I/O intensive tasks are made to be used effectively.

Now, consider how to deal with Reduce slots. We suspect that adding Reduce slots is not much effective for Hadoop jobs execution, because of the implementation of default Hadoop which avoid processing a lot of Reduce tasks simultaneously on a node. The job scheduler assigns only one Reduce task to

a TaskTracker every scheduling turn, and in the first place, the number of Reduce tasks which consist in a job predetermined to about 90% of the number of Reduce slots is kept fewer. Also, most of Hadoop users set the number of Reduce slots to more numerous number in the case of using many-core computers as slave nodes. However, the job scheduler takes at minimum the time of HeartBeat communication period times the number of Reduce slots to assign Reduce tasks to the full extent to such TaskTrackers because of the implementation such as described above. Consequently, it may be impossible to use added free Reduce slots. For these reasons, we do not handle the controlling the number of Reduce slots.

B. Implementation

We implemented our method on Hadoop 1.0.3 as follows:

- Initialization
  When MapReduce daemons single JobTracker and multiple TaskTrackers start, each TaskTracker reads the required values from the configuration file *mapred-site.xml* and initialize them. The initial number of Map slots predetermined as the property *mapred.tasktracker.map.tasks.maximum* is stored with the variable *maxMapSlots* and simultaneously with the variable *maxMapSlots_first* as the initial value in order to change *maxMapSlots* in our method. Users have been enabled to predetermine the threshold value for decision whether or not to add Map slots as the property *mapred.tasktracker.up.ioline* and the threshold value for decision whether or not to turn back Map slots as the property *mapred.tasktracker.down.ioline*. Each of these two values read from the configuration file at initialization is stored with the variable *up_ioline* or the variable *down_ioline*. Furthermore, all the arrays to be stored required information related to each CPU are initialized by using the number of CPUs read from /proc/stat on each slave node with active TaskTracker as the number of elements.
- Computing I/O wait percentage
  The way of computing I/O wait percentage has been based on mpstat command[10]. Both old and new value of I/O wait time spent and total CPU time spent, which are read from /proc/stat, are necessary to compute I/O wait percentage. The old values are computed just after calling the heartbeat method of JobTracker, in contrast, the new ones are just before it along with the sequence of steps. These values related to each CPU are acquired and maintained. Acquiring the new values, each TaskTracker computes current temporary I/O wait percentage with the new and old value, then determine the average of it and previous temporary one to final I/O wait percentage. Note that current temporary I/O wait percentage are stored with previous temporary I/O wait percentage after changing the number of Map slots.
- Controlling the number of Map slots
  The number of Map slots are controlled with computed recent final I/O wait percentage of each CPU. At first, in the case that a Map slot has not added owing a CPU, if I/O wait percentage of the CPU is greater than *up_ioline*, the variable *slot_balance* is incremented and the counter *back_count* of the CPU is set. the variable *slot_balance* is the difference from the initial number of Map slots, whose initial value is zero and maximum value is the number of CPUs of the slave node. the counter *back_count* of each CPU is used for turning back added Map slots. The number of Map slots can change drastically if the added Map slots are eliminated just after the detection of the lower rate of I/O wait than *down_ioline*. Hence, this counter has been provided and its initial value is set to two for the current implementation. Otherwise, in the case that I/O wait percentage of the CPU is lower than *down_ioline*, the *back_count* is decremented, then the *slot_balance* is decremented if the *back_count* has reached zero. Processing these steps related to each CPU, the number of Map slots is determined to *maxMapSlots_first* + *slot_balance* by using the method *setMapMapSlots*.

IV. Performance Evaluation

In order to make our method available, we have implemented it on Hadoop 1.0.3 and evaluated its performance in the environment shown in Table 1.

The threshold values for decision whether or not to control the number of slots have been determined based on the results of some preliminary experiments in the environment for this evaluation. When we have evaluated *up_ioline* and *down_ioline*, there is just the little difference between the results of the values which are close. Also, setting *up_ioline* to the extremely high values can cause the performance decrement because the high rate of I/O wait has appeared to a mound when the transition of I/O wait percentage has been indicated on graph. For this reason, *up_ioline* is set to 50 and *down_ioline* is set to 10.

We use *Sort*[1] program as benchmark program because *Sort* is basic operation as the program working on Hadoop MapReduce. The Map function *IdentityMapper* and the Reduce function *IdentityReducer* only get Key-Value pairs from RecordReader and output them. The amount of CPU processing in both of Map and Reduce phases is less than the amount of I/O processing, so that *Sort* is I/O intensive program.

The performance evaluation results with the clusters consist of 4, 8 or 16 slave nodes is shown in Fig.1. The graph compares default Hadoop and Hadoop implemented our method with the execution time of single job and each bar is normalized to the one of default Hadoop. Each slave node has two Map slots and two Reduce slots initially. The benchmark program is *Sort* and executed with the random data sets of key-value pairs generated by *RandomWriter*[1]. The sizes of them are 3GB, 6GB, 9GB, 12GB and 15GB. The execution time consists of three parts: Map, Map+Reduce and Reduce. Map is the state which is in only Map phase, Map+Reduce is the mixed state which is in both of Map and Reduce phases because Reduce phase starts before Map phase completes, and

Version	hadoop 1.0.3
File system	Hadoop File System
Benchmark program	Sort[1]
Scheduler	JobQueueTaskScheduler[1]
Master node	
CPU	(AMD Opteron 12-core 6168 / 1.9GHz) x2
OS	CentOS 5.7
Kernel	Linux 2.6.18
Memory size	32GB
Slave nodes	
CPU	Core i5 750 / 2.66GHz
OS	Ubuntu 11.4
Kernel	Linux 2.6.38
Memory size	8GB

TABLE I
PERFORMANCE ENVIRONMENT

Reduce is the state which is in only Reduce phase after Map phase completes. The effectiveness of our proposal is increased thanks to use the low usage CPU resources effectively in proportion to the scale of job (the size of its input data this time) for the scale of the cluster. In contrast, if the scale of job for the scale of the cluster is too smaller, the execution time of Hadoop implemented our method is longer than the one of default Hadoop. This is due to a transient state with the high rate of I/O wait of CPUs and the frequency of occurring task I/O wait is low in the first place.

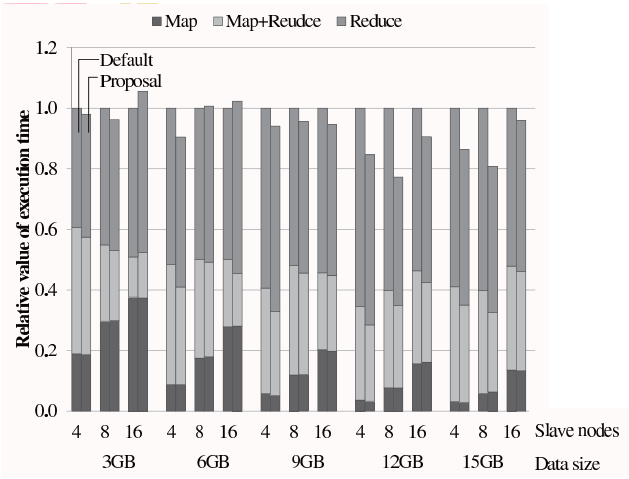

Fig. 1. Performance evaluation

Next, we have evaluated several times the fluctuation of the total number of Map slots in the whole cluster during the execution of single job with the cluster consists of 8 slave nodes and 12GB input data, because most effectiveness was found out in the execution pattern. The typical example of these results is shown in Fig.2. The number of the total initial number of Map slots in the whole cluster is set to 16, because this cluster consists of 8 slave nodes. The execution time of Map phase is about 50% of all execution time (including the state Map+Reduce) according to the analysis of it as shown in Fig.1. The total number of Map slots is increased up to 42 in the second half part. This results from a number of running I/O intensive Reduce tasks in this part. At this time, all Map tasks have completed. In the first half part of all experimental results, the total number of Map slots has increased up to 27 and about 20 on average.

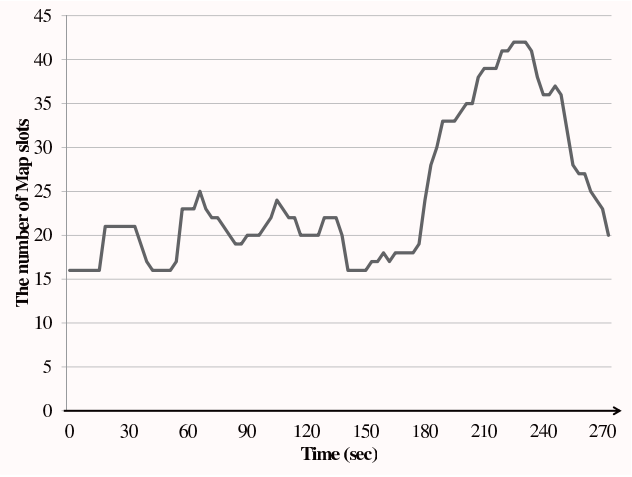

Fig. 2. One example of the fluctuation of the total number of Map slots during the execution of single job

Furthermore, it is shown in Fig.3 that the execution times of single job have been evaluated by using Hadoop implemented our method and default Hadoop with the varied total initial numbers of Map slots. The default value is set to 16, the average value of the first half part is set to 20 and the maximum value of the part is set to 27. *Sort* jobs have been executed with these four conditions. The results executed with both of the average and the maximum initial total number of Map slots show that their execution times are improved about 7% from the default value. However, we suspect that these values are only more suitable accidentally than the default in this environment. There are various theories as to the suitable number of Map slots related to the number of CPUs. It is difficult to determine the value appropriately, because it can change due not only to the number of CPUs of slave nodes but also to the execution environment and running jobs. Additionally, the execution times with both of the average and the maximum total initial number of Map slots are much the same, although the maximum value is 7 greater than the average value. Compared with these results, the execution time of Hadoop implemented our method has been much improved. This result shows that our proposed slots scheduling algorithm that controls the number of Map slots dynamically using I/O loads is effective in I/O intensive jobs.

V. CONCLUSION AND FUTURE WORK

In this paper, we proposed dynamic processing slots scheduling for I/O intensive job of Hadoop MapReduce to use the CPU resources with low usage effectively caused by I/O wait related to task execution which appears during executing job on Hadoop cluster. In order to examine the effectiveness

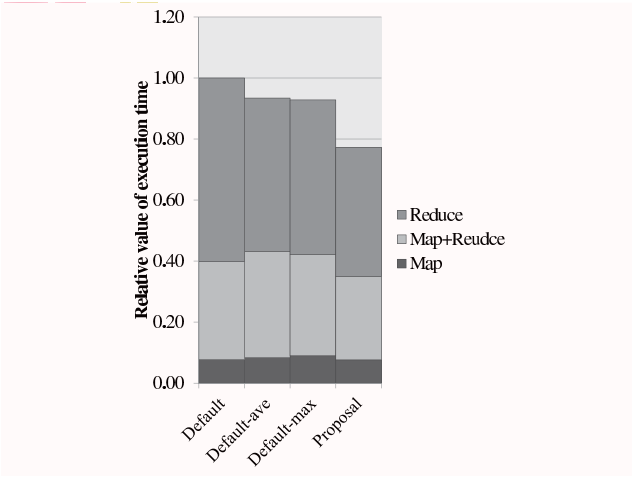

Fig. 3. The execution times with 8 slave nodes and 12GB input data

of our proposal, we implemented the our proposed method on Hadoop 1.0.3 and evaluated it by executing jobs with *Sort* as the benchmark program. Modified Hadoop was enabled to control the number of Map slots dynamically in comparison with default static assignment. Using our proposed method, the execution time was improved up to about 23% compared with default Hadoop.

There are some future works for our studies. One of these is switching how to control the number of Map slots according to the change of MapReduce phases. For example, in the case that there are only Reduce tasks in the job queues, adding Map slots is actually pointless. Also, the decision whether or not to perform the sequence of processes related to controlling the number of slots is necessary to avoid the overhead of managing threads to become greater than the effectiveness of adding Map slots.

By addressing these issues, we aim to develop the more polished faster of executing jobs originated by Hadoop.

ACKNOWLEDGEMENTS

This work was supported in part by KAKENHI, a Grant-in-Aid for Scientific Research (C),24500113.

REFERENCES

[1] *Hadoop*, The Apache Software Foundation, May 2012, 1.0.3.
[2] Apache hadoop. [Online]. Available: http://hadoop.apache.org/
[3] S. Ghemawat, H. Gobioff, and S.-T. Leung, "The google file system," in *19th ACM Symposium on Operating Systems Principles*, Lake George, NY, Oct. 2003.
[4] T. White, *Hadoop*, R. Tamagawa and S. Kaneda, Eds. O'Reilly Japan, 2010.
[5] J. Dean and S. Ghemawat, "Mapreduce: simplified data processing on large clusters," in *6th conference on Symposium on Opearting Systems Design and Implementation*, Berkeley, USA, Dec. 2004.
[6] S. Ibrahim, H. Jin, L. Lu, B. He, G. Antoniu, and S. Wu, "Maestro: Replica-aware map scheduling for mapreduce," in *12th IEEE/ACM International Symposium on Cluster, I Cloud and Grid Computing*, Ottawa, Canada, May 2012.
[7] X. Zhang, Y. Feng, S. Feng, J. Fan, and Z. Ming, "An effective data locality aware task scheduling method for mapreduce framework in heterogeneous environments," in *International Conference on Cloud and Service Computing*, Hong Kong, China, Dec. 2011.
[8] M. Hammoud and M. F. Sakr, "Locality-aware reduce task scheduling for mapreduce," in *3rd IEEE International Conference on Cloud Computing Technology and Science*, Athens, Greece, Nov./Dec. 2011.
[9] C. He, Y. Lu, and D. Swanson, "Matchmaking: A new mapreduce scheduling technique," in *3rd IEEE International Conference on Cloud Computing Technology and Science*, Athens, Greece, Nov./Dec. 2011.
[10] *sysstat*, The Linux Foundation, Jul. 2012, 10.1.1.