International Journal of Grid Distribution Computing
Vol. 8, No.5, (2015), pp.33-46
http://dx.doi.org/10.14257/ijgdc.2015.8.5.03

PCSP: A Preemptive Capacity Scheduler Policy for Scheduling Hadoop Jobs

Xue Shengjun[1], Wang Delong[2] and Shi Suhong[3]

[1,2,3]*School of Computer & Software, Nanjing University of Information Science & Technology, Nanjing 210044, China*
School of Computer & Software, Nanjing University of Information Science & Technology, Jiangsu Engineering Center of Network Monitoring, Nanjing 210044, China
School of Electronic & Information Engineering, Nanjing University of Information Science & Technology, Nanjing210044, China
[1]*sjxue@163.com*, [2]*king_de_long@126.com*, [3]*ssh1990518@163.com*

Abstract

Map reduce is a parallel programming paradigm used for processing massive data sets. A popular open-source implementation of Map reduce is Hadoop. There are basic schedulers embedded in Hadoop, including First in First out (FIFO), Fair Scheduler, and Capacity Scheduler (CS). Currently, researches have been focused on Capacity Scheduler to improve the Capacity Scheduler. Native Capacity Scheduler does not support the preemption, which results in the starvation caused by the non-preemptive scheduling. To resolve this problem, a Preemptive Capacity Scheduler Policy (PCSP) is proposed. Finally, we implement the PCSP on Hadoop, the experimental results of which indicate that PCSP we proposed is efficient in running Hadoop jobs.

Keywords*: PCSP, Capacity Scheduler, preemption, Hadoop, Map reduce*

1. Introduction

Cloud computing has gained increasing attention due to the fact that it can provide higher efficient computation. And it also provides flexible and scalable computing resources, such as computation power and data storage by organizing massive machines and clusters, to end-users. With the bursting of the data volumes, researches have been focused on Dryad [1] and Map reduces [2-5]. Hadoop [6], an open-source Map reduce implementation, has been widely adopted by industries such as Facebook and academia [7] It is a distributed programming model for parallel processing on TB level data sets on a large cluster consisting of thousands of machines. A key benefit of Map reduce is that it allows programmers to focus on developing software rather than dealing with issues, such as parallelization, scheduling, failover and input splitting. Map reduce becomes de-facto standards for processing large data sets in a cloud computing environment.

The Map reduce model mainly contains two phases: Map and Reduce. During the Map phase, the master node splits the input data into smaller data segments, and distributes them to several worker nodes for parallel processing by using a map function. The intermediate output produced by the map worker node is a collection of key-*value pair tuples*. During the Reduce phase, each reduce worker node gets the intermediate output by using the key and sorts them by checking the values and aggregates them by using a reduce function.

Since Hadoop jobs have to share the cluster resources, the scheduling policy, a critical factor for the performance, is used to determine when a job runsits tasks. The default scheduling policy of Hadoop is FIFO. The FIFO scheduler does not support preemption

ISSN: 2005-4262 IJGDC

based on priority. As a job with highpriority can still be blocked by a long-running job with low priority that started before the job with high priority is scheduled. FIFO neglects the demand differences among different users and different jobs without considering the actual load of Task Tracker. Besides FIFO, Fair Scheduler [8-9], which optionally supports preemption of jobs, developed at Facebook aims to give each job a fair share of the slot usage over time. The small job can obtain fast response, while large job can get guaranteed performance. In addition to the Fair Scheduler, the Capacity Scheduler [10] is developed by Yahoo, which restricts the number of slots occupied by the jobs requested by the same user and addresses a fair allocation of slots amongst users, especially in a large number of users. Compared with the Fair Scheduler, the Capacity Scheduler does not support the preemption, which means that a low-priority job can be temporarily swapped out to allow a higher-priority job to run.

Example 1. *User A submitted one job to Map reduce and User B also submitted one job with higher priority to Map reduce, which means User B's job is more urgent than User A's job. When User A's job is submitted for the first time, due to the non-preemption of Capacity Scheduler, the User B's job will be scheduled after Use A run its job*

In this paper, we first address the problem of Capacity Scheduler in detail on a Map reduce model. Then, we propose the improved policy based on the Capacity Scheduler that is utilized to realize the function of preemption. The rest of the paper is organized as follows. Section II is about the related work. Section III introduces outlines and technologies used in this work, including Hadoop, Map reduce and Capacity Scheduler. In section IV, we illustrate the disadvantage of Capacity Scheduler and present the improved algorithm. Section V implements the improved algorithm in Hadoop. Finally, a conclusion is given

2. Related Work

Ever since the advent of the Map reduce programming model, a huge number of studies have been dedicated to developing and improving Hadoop scheduler [11-12]. To our knowledge, a job scheduler is an essential component in each Hadoop system. As a result, it's necessary for us to pay more attention to job scheduler. A variety of Hadoop schedulers will be discussed in brief in the rest of this section.

K. Kc and K. Anyanwu [13] developed a deadline constraint scheduler to meet the maximum number of real-time jobs' deadline. What they proposed is a non-preemptive scheduler, which means that a job has to wait for other jobs' completion or half of the previous jobs that has been assigned to the slots. Compared with [13], Li Liu and Yuan Zhou*et al.* [14] proposed preemptive Hadoop jobs. The main principle of [14] is that the slot allocated to the map task is released and the output produced by this map task is sent to a reduce task if a map task is finished. *Deadline scheduler* was proposed in [15] by Xicheng Dong*et al*, who then developeda *two-level* scheduler that can schedule mixed real-time and non-real-time jobs according to their respective performance demands with resource preemption supported through integrating *Deadline scheduler* into existing Map reduce scheduler [16-17]. In [18], Fei Teng *et al.* proposed SPS scheduler, which supports job preemption with low context-switch overhead so that it can make an online scheduling decision when workflows randomly arrive in a cloud. Preemption is a significant technique for Map reduces schedulers to avoid delaying producing jobs and allowing the system can be shared by the other non-producing jobs. In addition, it also prevents a large job from occupying too many resources and starving the other jobs. However, Lu Cheng *et al.* [19] demonstrated that unnecessary preemption would degrade system normalized performance during busy periods and preserve fairness.

Among a variety of schedulers in Hadoop system, many schedulers in Hadoop schedule the tasks based on the priority. During the past few years, a lot of people used priority to improve the performance of scheduling jobs in Hadoop. Dynamic Priority

Scheduler [20] proposed by Sandholm *et al.* supports capacity distribution dynamically among concurrent users based on the priorities of the users. In [21], Phuong Nguyen and Tyler Simon *et al.* proposed a Hybrid Scheduler (HybS) algorithm based on dynamic priority for reducing the latency for variable length concurrent jobs and maintaining data locality. Peng Lu*et al.* [22] focused on the slot configuration, which has a significant impact on performance. It is demonstrated that high priority jobs sharing resources with low priority jobs leads to lower resource utilization, and presents a non-intrusive slot layering solution. Finally, a layering-aware task scheduler was developed to deal with these problems.

In summary, they do not consider preemptive scheduler based on priority. By the in-depth analysis of Capacity Scheduler, it is significant to meet the demand of preemption. However, the current Capacity Scheduler that does not support preemption can support priority by hand-configuring. In order to combine advantages of both preemption and priority, the preemptive Capacity Scheduler is proposed.

3. Preliminary Knowledge

3.1. Map Reduce

Hadoop is the most popular open-source implementation of Map reduces. The execution runtime includes partitioning the input data, scheduling the program's execution, dealing with machine failures and managing inter-machine communication. All of them are crucial in a distributed environment. In this paper, we present our scheduler prototype on top of Hadoop.

Map reduce, a programming model and an associated implementation for processing and generating large data sets, is widely used by researchers. Users specify a map function that processes a key-value pair to generate a set of intermediate key-value pairs and a reduce function that merges all intermediate values associated with the same intermediate key [23]. Each cluster-node of the Map reduce framework consists of a single master JobTracker and a slave TaskTracker per cluster-node. The master is responsible for scheduling the jobs component tasks on the slaves, monitoring them and re-running the failed tasks. The slaves follow the instruction of the master to execute the tasks. The job tracker includes a task scheduler module to assign tasks to different task trackers that periodically sends a heartbeat to the job tracker. The job checks the heartbeat and assigns tasks to available task trackers. The scheduler assigns each task to a node randomly via the same heartbeat message protocol. Figure 2 illustrates the flow chart of running a job.

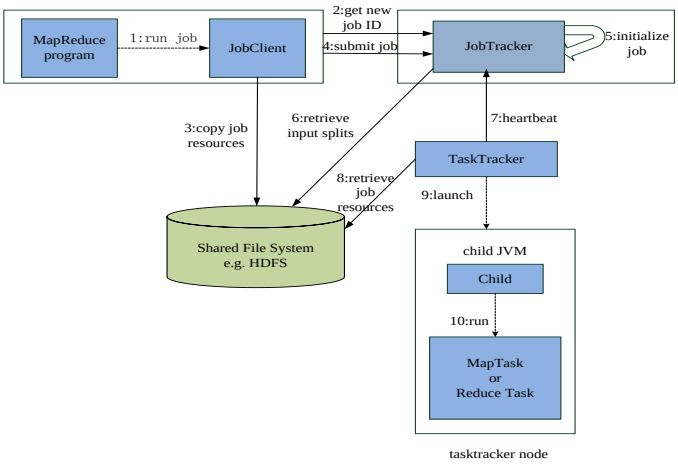

Figure 1. Map Reduce Operation Procedure [24]

3.2. Capacity Scheduler

In the Capacity Scheduler, there are three granularity objects, namely: queue, job and task. When an idle slot appears on a TaskTracker, the scheduler will select a queue, a job in this queue and the tasks of the selected job. And then assign the slot to the tasks. The following describes the selection strategy for queue, job and task [25]:

1) Select a queue: All queues are put in ascending order based on resource utilization so that you can easily choose.

2) Select a job: In the selected queue, all jobs are sorted according to the submission time and the priority.

3) Select a task: locality and resource utilization of this task will be taken into consideration in this stage.

The Capacity Scheduler works according to the principles following:

1) The existing configuration is obtained from the capacity - scheduler. xml at cluster startup.

2) An initialization Poller thread is also initialized. A single thread can handle multiple queues. All the jobs admitted into the system are not initialized instantly to reduce the memory footprints on the job tracker.

3) After a job is submitted to the cluster, the scheduler checks the job submission limits to determine if the job can be accepted or not (based on queue and user).

4) If the job can be accepted, the Poller will check the initialization limits for the job. If the job can be initialized, it would be submitted to the assigned worker thread in the queue that is responsible for initializing the job.

5) When the job-tracker gets the heartbeat from a task-tracker, a queue is selected from all the queues. A queue is selected by sorting the queues according to the number of running tasks/capacity of the queue. After selecting the queue, the first job is chosen from the queue unless its user is over the user limit. Then, a task is picked up and the preference is given to node-local task over rack-local task. This procedure is repeated until all jobs completed

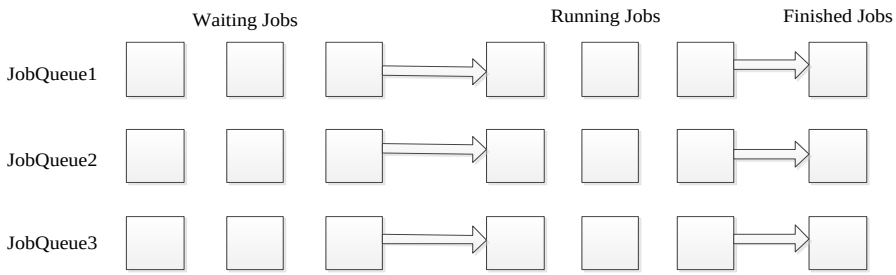

Figure 2. Scheduling Model of the Capacity Scheduler

4. Design and Implementation

4.1. Math Model Design

Multi-user and multi-queue mechanism is supported by Capacity Scheduler. Therefore, we first define a set of queues expressed as $Queues = \{Q_1, Q_2, \ldots, Q_m\}$, where m is less than that configured in the configuration file. And for each in $Queues$, some jobs $Q_j (0 < j < m)$ are contained. Parameter Q refers to the queue in the. Then $Q = \{J_1, J_2, \ldots, J_N\}$, where the job J_i with an input data size σ_i and its user, refers to the queue contains some jobs. For each queue in an array $Queues$, scheduling process is the same as the others. (Multi-user and multi-queue mechanism is supported by Capacity Scheduler. Therefore, we first

define a set of queues (expressed as) $Queues = \{Q_1, Q_2, \ldots, Q_m\}$, the value of m being less than that configured in the configuration file and parameter Q refers to the queue in the $Queues$. Each $Q_j (0 < j < m)$ in $Queues$ corresponds to some jobs, that is, $Q = \{J_1, J_2, \ldots, J_N\}$ where the job J_i has input data size σ_i and its user. The scheduling process for each queue in the array $Queues$ is the same. Table 1 shows the parameters we needed.

Table 1. The Abbreviation Parameters

Resource Usage	*ru*
numSlotsOccupied	*nso*
capacity	*cp*
currentCapacity	*cc*
slotsPerTask	*spt*
userLimit	*ul*
numJobsByUser	*njbu*
minValueOfUserLimit	*mvoul*
freeMemOnTT	*fmot*
totalMemUsedOnTT	*tmuot*
memUsedOnTT	*muot*

In our scheduling model, we assume that: (i) all machines are homogeneous; (ii) no failure occurs during scheduling and execution. The job initialization operation manager, referred to the *JobInitializationPoller*, is responsible for providing the initialization for every queue in the Capacity Scheduler. Among queues, they need to be sorted by *resourceUsage* defined as *ru*. The queue's priority is determined by the idle degree of the queues. That is, the more idle of the queue, the higher of its priority. In order to avoid wasting a lot of cluster computing resources, *JobInitializationPoller* adopts round-robin algorithm to choose appropriate counts of jobs (instance of *JobInProgress*), which must satisfied with two conditions: (i)the number of existing jobs of the user whom the current job belongs to is less than *maxJobsPerUserToInitialize*; (ii) the number of the same user is less than *maxUsersToInitialize*.

$$ru = nso / cp \tag{1}$$

When a job has been chosen successfully, it would be added to the *waitingJobs* list where jobs wait for initializing. After a job finishes its initializing process, it joins *runningJobs*. No matter the jobs are in *waitingJobs* or in *runningJobs*, both of them are sorted by submission time or priority. If the Capacity Scheduler works based on priority, jobs will be sorted according to the priority.

If one job in the *runningJobs* is expected to be chosen to be scheduled, it must satisfy two conditions: (i) capacity resource occupied by the current user who is the owner of the current job is less than *userLimit*. (ii) *freeMemOnTT*, which means free memory on TaskTracker, is enough for this job. How to calculate *userLimit*and *freeMemOnTT?* Firstly, if we are running below capacit

$$\text{cc} = cp \tag{2}$$

If we're running beyond capacity

$$\text{cc} = nso + spt \tag{3}$$

Finally,

$$ul = Max(cc / njbu, mvoul * cc / 100) \tag{4}$$

The method of calculating the *freeMemOnTT* will be discussed. *memUsedOnTT* that means memory has been used on TaskTracker is defined. Similarly, *totalMemUsedOnTT*

is defined.

$$fmot = tmuot - muot \tag{5}$$

When a job has been scheduled successfully, it would be divided into many tasks. At the same time, these tasks are distributed into TaskTracker by JobTracker with the consideration of locality and resource utilization.

It is well known that each job has priority that needs to be set in advance by hand. The values of priority are of the following types: VERY_HIGH, HIGH, NOMAL, LOW and VERY_LOW. High priority jobs have shorter execution time than lower priority jobs, depending on the priority type [17]. We need to quantify the value of priority. Table 2 shows the new value of the priority after quantifying the value of priority. In the native Capacity Scheduler, a list will be defined to store jobs and corresponding priorities. Sequentially, *schedulingSuccessfullyJobs* will be used to refer to the jobs scheduled successfully.

A critical problem will be discussed, that is, the circumstances of the preemption occurring. The preemption happens when priority of the first job waiting for being scheduled in the *runningJobs* is higher than it of the job scheduled successfully. Whenever a running Job among *runningJobs* is successfully scheduled the running Job would be deleted from *runningJobs* list and added to *schedulingSuccessfullyJobs*. When job1 in *runningJobs* set has a higher priority than job2 in *schedulingSuccessfullyJobs*, the job2 would be killed and stored in *runningJobs* list. If the job1's user utilized capacity resource doesn't exceed the *userLimit*and*freeMemOnTT*, this case meets the requirement of job1, and job1 will be scheduled. At the same time, Job1 is added to *schedulingSuccessfullyJobs* list. After job1 completed, job2 continue to be scheduled.

Table 2. The New Value of Priority

Priority	Value
VERY_HIGH	5
HIGH	4
NORMAL	3
LOW	2
VERY_LOW	1

4.2. PCSP: Preemptive Capacity Scheduler Policy

We have implemented a Preemptive Capacity Scheduler Policy (PCSP) as a pluggable scheduler in Hadoop system. The PCSP is designed to meet the two requirements: (i) User having important jobs to deal with will obtain the results firstly by giving higher priority than the running jobs; (ii) it overcomes the limitation of the Capacity Scheduler.

When a new job is submitted, the PCSP first decides whether the priority of this job is higher than that of the running job. If yes, the new job is allowed to be scheduled by the PCSP.

The PCSP executes in the following phases periodically.

(1) Initial Phase: PCSP collects necessary information from the configuration files called capacity-scheduler.xml, including Q.capacity, Q.support-priority, minimum-user-limit-percent, init-worker-threads, and recalculate the *userLimit*.

(2) Scheduling Phase: For all queues, *resourceUsage* is calculated for sorting the queues in ascending order. As discussed before, idle queues have advantages. For each queue contains many jobs, the fact whether the PCSP supports priority is confirmed. If yes, sorting jobs, according to priority is available. Otherwise,

startTime will be used for sorting jobs. After sorting the jobs, the job satisfying these two conditions is chosen to add the *waritingJobs* list for being initialized. After the job is initialized, the job would be added to the *runningJobs* for be scheduled.

(3) Preemption Phase: After the job is scheduled successfully, the job will be added to *schedulingSuccessfullyJobs* list. When a new job enters the *runningJobs* list, its priority would be checked. Only the job is satisfied with these two conditions, the preemption would occur.

(4) Allocated Phase: the PCSP allocates the slots to the tasks according to the results of scheduling and preemption phase. When *numSlotsOccupied* decreases to 0, the task suspends.

Algorithm: PCSP(Preemptive Capacity Scheduler Policy)

```
//Initial Phase
1: Collect Q.capacity, Q.support-priority.
//Scheduling Phase
2: Queues={Q1 ,Q2 ,…,Qm}
3: resourceUsage←numSlotOccupied/capacity
4: Sort Queues by resourceUseage
5: for 1 to m do
6:     ifQ.support-priority then
7:       Sort jobs by priority
8:     else then
9:       Sort jobs by startTime
10:    end if
11:    for job∈Q do
12:       if !isOverUserLimit and job.reqMemthen
13:          waitingJobs←job
14:          after initializing the job
15:          runningJobs←job
//Preemption Phase
16:          after scheduling the job successfully
17:          schedulingSuccessfullyJobs←job
18:          while runningJob∈runningJobs repeat
19:               if runningJob.priority>schedulingSuccessfullyJob.priority
20:                  and the job satisfies the two conditions discussed before
then
21:                  kill the schedulingSuccessfullyJob
22:                  add schedulingSuccessfullyJob to runningJobs
23:                  add runningJob to schedulingSuccessfullyJobs
24:               end if
25:            endwhile
26:       end if
27:       ifTaskType==MAP and memForthisTask<=freeMemOnTTthen
28:          scheduler←MapScheluer
29:       else TaskType ==REDUCE and memForthisTask<=freeMemOnTT
30:          scheduler←ReduceScheluer
31:       end if
32:    end for
33:end for
//Allocation Phase
34: Allocate and suspend the tasks according to numSlotsOccupied.
```

5. Experimental Evaluation

In this section, we describe some experiments that evaluating effectiveness of these scheduling algorithms: Capacity Scheduler, and PCSP. We discussed the method of

having our experiments, our experiments itself and the results.

5.1. Evaluation Methodology

We have experiments on an isolated 6-node cluster. One node is designated as the job-tracker and name-node. All the nodes are configured as follows. Measurements directly on hardware allows us to guarantee the consistency of the configuration on multiple nodes. Therefore, accurate experimental results will be promised. Table 3 shows the configuration of all the nodes. Besides, some Hadoop parameters must be configured in advance, the others by default. Table 4 shows the important Hadoop parameters.

Table 3. The Configuration of All the Nodes

Property	Value
CPU	Intel(R) Pentium(R) CPU @ 3.00 GHZ
Memory	1 GB
Operation System	Centos 6.4
Hadoop Version	hadoop 1.2.1
Hard Disk	100 GB

Table 4. The Configuration of Important Hadoop Parameters

Hadoop Parameter	Value
dfs.replication	2
dfs.block.size	64M
dfs.heartbeat.interval	5s
speculative task	Enabled

WordCount was used as a benchmark Map reduces application for the experiments. The size of the input data file is 1 GB in the executed Map reduces application. As mentioned before, the number of the TaskTrackers is 5 and there are 10 maps and 10 reduce slots (i.e. 2 slots per node for each phase). In our experiments, we use the average waiting time of the jobs with higher priority and completion time as the performance metric. The average waiting time of the jobs with higher priority means how long the jobs with higher priority will wait for running with low priority to be killed? The execution time includes time from submission to the completion time. The time shown in our results is an average of 3 runs. Before conducting the experiments, we must guarantee that the function of scheduling the jobs based on priority is supported. Multiple users were supported to start jobs at the same time.

5.2. Results

Under these experiments environments, the number of queues is always 2. Each queue has 3 users and each of them submits one job, one after another at the interval of one second. All the users in each queue submit application *WordCount*. Users 1-3 are in queue 1, and users 4-6 are in queue 2 in all the results shown in the paper. Each job must be set priority before scheduling the jobs. Subsequent experiments use the same settings in terms of the number of users per queue and their job submission pattern, unless stated otherwise. Table 5 shows the related information of each job in detail.

Table 5. Detail Information In The Case Of Multiple Queues and Multiple Users

Job ID	Queue	User	Priority Value
Job1	queue1	user1	NORMAL
Job2	queue1	user2	NORMAL
Job3	queue1	user3	HIGH
Job4	queue2	user4	LOW
Job5	queue2	user5	NORMAL
Job6	queue2	user6	VERY_LOW

After starting Hadoop, the jobs were submitted to the PCSP one after another, and all the jobs were set different initial priority. Recording job completion sequence and the results were compared with the Capacity Scheduler scheduling algorithm. As for Capacity Scheduler, the job was submitted one by one at the interval of one second. Hence, the initialization and execution of jobs were based on the priority value. Compare with the Capacity Scheduler, the job1 and job2 were submitted after one minute in queue1. If job1, job2, and job3 are submitted at the same time, we will not get the correct experimental results according to the principle of Capacity Scheduler. To avoid this phenomenon, we need to do what I formulated earlier. Table 6 shows the test results.

Table 6. Test Results

Scheduling Algorithm	Capacity Scheduler	PCSP
Test Results	Job3, Job1,Job2, Job5,Job4,Job6	Job1, Job3,Job2, Job5,Job4,Job6

Exactly as the rule of submitting the jobs, when a job is running in Hadoop platform, one job with higher priority is available for preempting the map slots occupied by running the job. It proves to be that PCSP supports the preemption in the case of multiple queues and multiple users, which is the advantage of the previous Capacity Scheduler. We have another experiment on single queue and multiple users and Table 7 shows the related information of each job in detail.

Table 7. Detail Information in the Case of Single Queue and Multiple Users

Job ID	Queue	User	Priority Value
Job1	queue1	user1	VERY_HIGH
Job2	queue1	user2	NORMAL
Job3	queue1	user3	HIGH

Table 8. Test Results

Scheduling Algorithm	Capacity Scheduler	PCSP
Test Results	Job1, Job3, Job2	Job1, Job3,Job2

The same method was used to conduct this experiment. Coincidentally, the results of the Capacity Scheduler and PCSP are the same. But Table 10 shows the different results compared with Table 8. When job1, job2, and job3 were submitted to the Capacity Scheduler, they were sorted by priority at the beginning. Therefore, the result of the Capacity Scheduler is accurate. As for PCSP, where Job1 and Job2 were submitted, owing to the Job1 with higher priority, Job1 was scheduled first, where it won't bring out preemption. When Job1 was finished, the Job3 was submitted, obviously Job2 was

running. However, Job3 possessed the higher priority, leading to the occurrence of preemption. Job2 was killed and then Job3 occupied the map slots and capacity resources. Finally, Job2 was waiting for the Job3 to be finished. Therefore, Job3 was the last one to be finished. The same analysis method applies to the Table 10.

Table 9. Detail Information in the Case of Single Queue and Multiple Users

Job ID	Queue	User	Priority Value
Job1	queue1	user1	LOW
Job2	queue1	user2	NORMAL
Job3	queue1	user3	HIGH

Table 10. Test Results

Scheduling Algorithm	Capacity Scheduler	PCSP
Test Results	Job3, Job2, Job1	Job2, Job3,Job1

Improved Capacity Scheduler was proved to be preemption-supported based on the priority by the experimental results. The difference between Capacity Scheduler and PCSP execution time would be discussed later.

WordCount was used to obtain the execution time of Capacity Scheduler and PCSP. Their input size was 20GB. Due to the block size, the number of map tasks was 16.The classic program *WordCount* was submitted to the Capacity Scheduler and the PCSP respectively. Each job was executed ten times for calculating the average execution time to make the experimental results more accurate. Table 11 shows the information about this experiment.

Table 11. Related Information about This Experiment

JobID	Queue	User	Priority	Program
Job1	queue1	user1	NORMAL	*WordCount*
Job2	queue1	user2	NORMAL	*WordCount*
Job3	queue1	user3	HIGH	*WordCount*
Job4	queue2	user4	LOW	*WordCount*
Job5	queue2	user5	NORMAL	*WordCount*
Job6	queue2	user6	VERY_LOW	*WordCount*

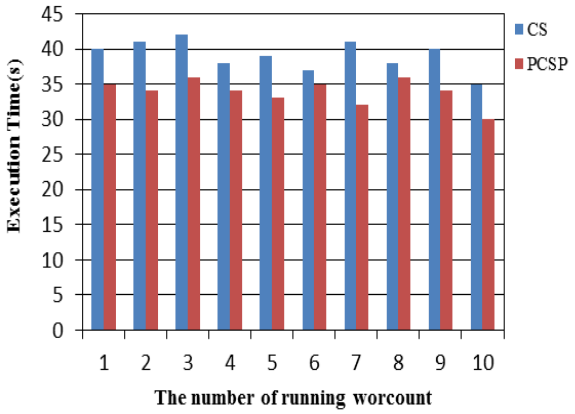

Figure 3. Execution time of CS with that of PCSP

According to the experimental result of Figure 3, although preemption consumed time, such as killing job, removing a job from *schedulingSuccessfully* to *runningJob*, and deleting a job in *schedulingSuccessfullyJobs* list, however, the cost of preemption can be neglected in fact. The execution time of the PCSP was less than the Capacity Scheduler for each time. The average time of PCSP was also less than the Capacity Scheduler. Therefore, PCSP was efficient in this case.

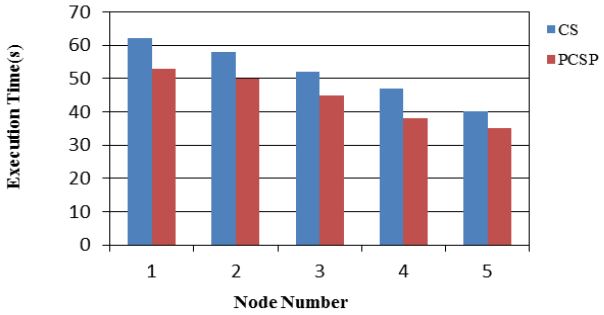

Figure 4. Execution Time of CS with that of PCSP with Different Node Number

Different number of nodes was used to find out the differences between Capacity Scheduler and PCSP in terms of time cost when scheduling *WordCount* job. As is shown in Figure 4, with the increment of the number of nodes, PCSP's time cost is more efficient than Capacity Scheduler's.

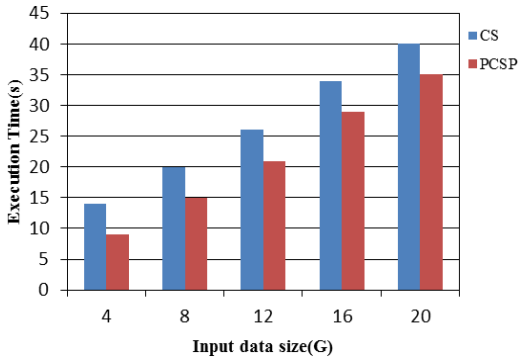

Figure 5. Execution Time of CS with that of PCSP with Different Data Size

Different input data size was used to find out the differences between Capacity Scheduler and PCSP time cost when scheduling *WordCount* jobs. Under the premise of the same number of nodes, input data size was respectively set up as 4GB, 8G, 12G, 16G, 20G. As is shown in Figure 5, with the increase of input data size, PCSP is more efficient than Capacity Scheduler in terms of time cost.

6. Conclusions

In this paper, we employed PCSP in Hadoop to overcome the advantage of Capacity Scheduler, such as the starvation caused by the non-preemptive scheduling. We presented the problem, proposed the model of the Preemptive Capacity Scheduler, and implemented it on a real Map reduce system—Hadoop. Our experiment shows that: (1) the PCSP gets better performance of running time than Capacity Scheduler when scheduling the jobs having computational intensive map and reduce tasks. (2) The PCSP overcomes the issue

of Capacity Scheduler observed from the experimental results. In our future work, we will consider the aspect that can have effects on the performance of scheduling, including data distributions in a heterogeneous Map reduce system, iterative Map reduce jobs. Besides, we also delve into the load balance of all the nodes based on PCSP.

Acknowledgment

This work was partially supported by the National Natural Science Foundation of China (Grant Nos.41275116), Jiangsu Economic and Information Technology Commission project of China(Grant Nos.{2011}1178) and Open Fund Project of Jiangsu Engineering Center of Network Monitoring (Grant Nos.KJR1309).

References

[1] M. Isard, M. Budiu, Y. Yu, A. Birrell and D. Fetterly, "Dryad, distributed data-parallel programs from sequential building block", ACM SIGOPS Operating Systems Review, vol. 41, no. 3, **(2007)**, pp. 59-72.
[2] J. Dean and S. Ghemawat, "Map reduce: simplified data processing on large clusters", Communications of the ACM, vol. 51, no. 1, **(2008)**, pp. 107-113.
[3] L. Wang, J. Tao, R. Ranjan, H. Marten, A. Streit, J. Chen and D Chen, "G-Hadoop, Map reduce across distributed data centers for data-intensive computing", Future Generation Computer Systems, vol. 29, no. 3, **(2013)**, pp.739-750.
[4] J. Tan, X. Meng and L. Zhang, "Delay tails in Map reduce scheduling", ACM SIGMETRICS Performance Evaluation Review, vol. 40, no. 1, **(2012)**, pp. 5-16.
[5] H. Kang, Y. Chen, J. L. Wong, R. Sion and J. Wu, "Enhancement of Xen's scheduler for Map reduce workloads", In Proceedings of the 20th international symposium on High performance distributed computing, **(2011)**, pp.251-262.
[6] Y. Lee and Y. Lee, "Toward scalable internet traffic measurement and analysis with Hadoop", ACM SIGCOMM Computer Communication Review, vol. 43, no. 1, **(2011)**, pp. 5-13.
[7] S. Ibrahim, H. Jin, L. Lu, B. He, G. Antoniu and S. Wu, "Maestro: Replica-aware map scheduling for Map reduce", Cluster, Cloud and Grid Computing (CCGrid), 2012 12th IEEE/ACM International Symposium, **(2012)**, pp. 435-442.
[8] D. Yoo and K. M. Sim, "A comparative review of job scheduling for Map reduce", Cloud Computing and Intelligence Systems (CCIS), 2011 IEEE International Conference on. IEEE, **(2011)**, pp. 353-358.
[9] M. Zaharia, D. Borthakur, J. S. Sarma, K. Elmeleegy, S. Shenker and StoiacaI, "Job scheduling for multi-user Map reduce clusters", EECS Department, University of California, Berkeley, Tech. Rep. USB/EECS-2009-55, **(2009)**.
[10] "Capacity Scheduler Guide", the Apache Software Foundation, User Manual, **(2013)**.
[11] T. G. Addair, D. A. Dodge, W. R. Walter and S. D. Ruppet, "Large-scale seismic signal analysis with Hadoop", Computers & Geosciences, **(2013)**, pp.145-154.
[12] Y. Lee and Y. Lee, "Toward scalable internet traffic measurement and analysis with hadoop", ACM SIGCOMM Computer Communication Review, vol. 43, no. 1, **(2013)**, pp. 5-13.
[13] K. Kc and K. Anyanwu, "Scheduling hadoop jobs to meet deadlines", Cloud Computing Technology and Science (CloudCom), 2010 IEEE Second International Conference on. IEEE, **(2010)**, pp. 388-392.
[14] L. Liu, Y. Zhou, M. Liu, G. Xu, X. Chen, D. Fan and Q. Wang, "Preemptive Hadoop Jobs Scheduling under a Deadline", Semantics, Knowledge and Grids (SKG), 2012 Eighth International Conference on. IEEE, **(2012)**, pp. 72-79.
[15] X. Dong, Y. Wang and H. Liao, "Scheduling mixed real-time and non-real-time applications in Map reduce Environment", Parallel and Distributed Systems (ICPADS), 2011 IEEE 17th International Conference on. IEEE, **(2011)**, pp. 9-16.
[16] F. Li, B. C. Ooi, M. T. Ozsu and S. Wu, "Distributed data management using Map reduce", ACM Computing Survey, **(2013)**.
[17] B. Aksanli, J. Venkatesh, L. Zhang and T. Rosing, "Utilizing green energy prediction to schedule mixed batch and service jobs in data centers", ACM SIGOPS Operating Systems Review, vol. 45, no. 3, **(2012)**, pp. 53-57.
[18] F. Teng, H. Yang, T. Li, Y. Yang and Z. Li, "Scheduling real-time workflow on Map reduce-based cloud", Innovative Computing Technology (INTECH), 2013 Third International Conference on IEEE, **(2013)**, pp. 117-122.
[19] L. Cheng, Q. Zhang, R. Boutaba, "Mitigating the negative impact of preemption on heterogeneous Map reduce workloads", Proceedings of the 7th International Conference on Network and Services Management. International Federation for Information Processing, **(2011)**, pp. 189-197.
[20] T. Sandholm and K. Lai, "Dynamic proportional share scheduling in Hadoop", Job scheduling strategies

for parallel processing. Springer Berlin Heidelberg, **(2010)**, pp.110-131.
[21] P. Nguyen, T. Simon, M. Halem, D. Chapman and Q. Le, "A Hybrid Scheduling Algorithm for Data Intensive Workloads in a Map reduce Environment", Utility and Cloud Computing (UCC), 2012 IEEE Fifth International Conference on. IEEE, **(2012)**, pp.161-167.
[22] P. Lu, Y. C. Lee, A. Y. Zomaya, "Non-intrusive Slot Layering in Hadoop", Cluster, Cloud and Grid Computing (CCGrid), 2013 13th IEEE/ACM International Symposium on. IEEE, **(2013)**, pp. 253-260.
[23] Y. Fengying, L. Huichao, Z. Zhangping, "Research on Cloud-Based Mass Log Data Management Mechanism", Journal of Computer, vol. 9, no. 6, **(2014)**, pp. 1371-1377.
[24] Y. Lin and G. Xiangyao, "Parallel Clone Code Detector in Map reduce", Journal of Software, vol. 9, no. 6, **(2014)**, pp. 1561-1566.
[25] H. Nordberg, K. Bhatia, K. Wang and Z. Wang, "BioPig, a Hadoop-based analytic toolkit for large-scale sequence data", Bioinformatics, vol. 29, no. 23, **(2013)**, pp. 3014-3019.

