A Dynamic MapReduce: Scheduling in Large Cluster

[1]Iniyan Senthil Kumar.T, [2]Poovaraghan.R.J

[1]student, [2]Assistant Professor
[1]Computer Science and Engineering
[1]SRM University, Chennai, India

Abstract: **MapReduce is the program model and an association implementation for processing and generating large data sets. User specify a map function that process a key/value pair to generate a set of intermediate key/value pairs and a reduce function that merge all intermediate values associated with the same intermediate key. MapReduce is important Programming model for building data centres containing ten thousands of nodes. In a Practical data, I/O bound jobs and CPU-bounds jobs, which demand different resources, run simultaneously in the same cluster. In this Paper we gives a new view of the MapReduce model and classify the MapReduce workload into three categories based on their CPU and I/O Utilization. We Design a new dynamic MapReduce workload predict mechanism, MR-Predict mechanism, which detect the workload types on the fly. We propose a Triple-Queue scheduler based on the MR-Predict mechanism. The Triple-Queue could improve the usage of both CPU and disk I/O resources under heterogeneous workload.**

IndexTerms **- MapReduce, Hadoop, Cluster, Schedule, heterogeneous workload**

I. Introduction

As the Internet scale keeps growing up, enormous data needs to be processed in many Internet Service Providers.
MapReduce framework is now becoming a leading example solution for this. MapReduce is designed for building large commodity cluster, which consists of thousands of nodes by using commodity hardware. Hadoop, a popular open source implementation of MapReduce framework, developed primarily by Yahoo, is already used for processing hundreds of terabytes of data on at least 10,000 cores [3]. In this environment, many people share the same cluster for different purpose. The term "Big Data" refers to large and complex data sets made up of a variety of structured and unstructured data which are too big, too fast, or too hard to be managed by traditional techniques.
In this work, we concentrate on the problem that how to improve the hardware utilization rate when different kinds of workloads run on the clusters in MapReduce framework. In practical, different kinds of jobs often simultaneously run in the data centre. These different jobs make different workloads on the cluster, including the I/O-bound and CPU-bound workloads. But currently, the characters of workloads are not aware by Hadoop's scheduler which prefers to simultaneously
run map tasks from the same job on the top of queue
We design a new triple-queue scheduler which consist of a workload predict mechanism MR-Predict and three different queues (CPU-bound queue, I/O-bound queue and wait queue).
We classify MapReduce workloads into three types, and our workload predict mechanism automatically predicts the class of a new coming job based on this classification. Jobs in the CPU bound queue or I/O-bound queue are assigned separately to parallel different type of workloads. Our experiments show that our approach could increase the system throughput up to 30% in the situation of co-exiting diverse workloads

II. MapReduce Overview

MapReduce is a programming paradigm for processing large data sets in distributed environments [3]. In the MapReduce paradigm, the *Map* function performs filtering and sorting, while the *Reduce* function carries out grouping and aggregation operations. The 'hello world' of MapReduce is the word counting example: it counts the appearance of each word in a set of documents. The *Map* function splits the document into words and for each word in a document it produces a (key, value) pair.

```
function map(name, document)
for each word in document
   emit (word, 1)
```

The *Reduce* function is responsible for aggregating information received from *Map* functions. For each key, word, the *Reduce* function works on the list of values, partial Counts. To calculate the occurrence of each word, the *Reduce* function groups by word and sums the values received in the partialCounts list.

```
function reduce (word, List partialCounts)
  sum = 0
  for each pc in partialCounts
    sum += pc
  emit (word, sum)
```

The final output is the list of words with the count of appearance of each word.
Figure 1 illustrates the MapReduce flow. One node is elected to be the master responsible for assigning the work, while the rest are workers. The input data is divided into splits and the master assigns splits to *Map* workers. Each worker processes the

corresponding input split, generates key/value pairs and writes them to intermediate files (on disk or in memory). The master notifies the *Reduce* workers about the location of the intermediate files and the *Reduce* workers read data, process it according to the *Reduce* function, and finally, write data to output files.

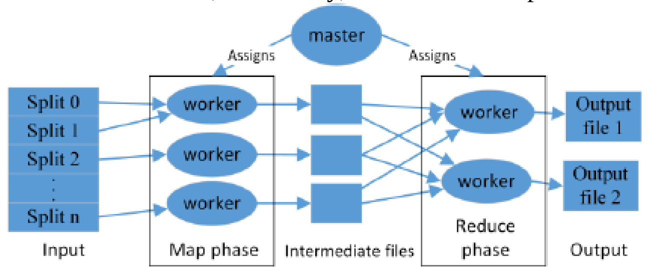


Figure 1. MapReduce flow

The main contribution of the MapReduce paradigm is scalability as it allows for highly parallelized and distributed execution over a large number of nodes. In the MapReduce paradigm, the *Map* or *Reduce* task is divided into a high number of jobs which are assigned to nodes in the network. Reliability is achieved by reassigning any failed node's job to another node. A well-known open source MapReduce implementation is Hadoop which implements MapReduce on top of the Hadoop Distributed File System (HDFS).

III. MAPREDUCE ANALYSIS AND WORKLOAD

In this section, we analyze the MapReduce job working procedure, and give a classification of workloads on MapReduce. Then we discuss the schedule model in Hadoop, and express the problem on this model when diverse workloads run on the current implementation of Hadoop.
.

A.MapReduce Procedure Analysis:

MapReduce contains a map phase grouping data in specified key and a reduce phase aggregating data shuffled from map nodes. Map tasks are a bag of independent tasks which use different input. They are assigned to different nodes in cluster. In the other hand, reduce tasks depend on the output of map tasks. They keep fetching map result data from other nodes. Shuffle actions which are I/O-bound often intercross in the map phase, for maximizing the utility of I/O resource. As shown in Figure 1, after all needed intermediate data get shuffled, the reducer begins to compute.

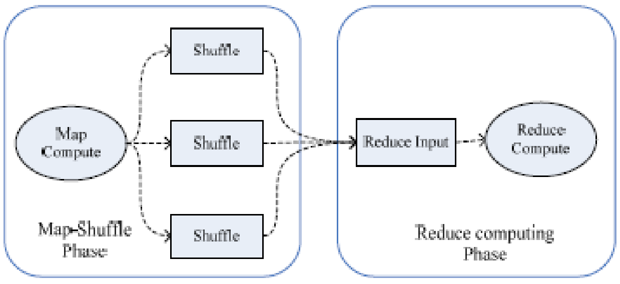


Figure 2.MapReduce Data Process Phases

We decompose the MapReduce procedure into two sub phases which are Map-Shuffle phase and Reduce-Computing phase. In the Map-Shuffle phase, every node dose five actions:
1) init input data; 2) compute map task; 3) store output result to local disk; 4) shuffle map tasks result data out; and 5) shuffle reduce input data in. The Map-Shuffle phase is the first step. And in Reduce-Computing phase, tasks could directly begin to run the application logic because the input data is already shuffled in memory or local disk.
In this view of MapReduce, Map-Shuffle phase is critical to the whole procedure. In this phase, every node performs their map task logic which is similar in one job, and shuffles result data to all reducer nodes. All reducer nodes are not able to begin computing because just one map node slows down. Our work focuses on this phase in MapReduce. We predict map tasks behaviour by analysing job's Map-Shuffle phase history. I/O-bound map tasks and CPU-bound map tasks could be distinguished by our scheduler. Then we parallel the different kinds of workloads.

B.Classification on Workload

According to the utilization of I/O and CPU, we give a classification of workloads on the Map-Shuffle phase of MapReduce. As we say, every node in the Map-Shuffle phase does five actions
The ratio of the amount of map input data (MID) and map output data (MOD) in a single map task depends on the type of workload. We define a variable ρ as the application logic of particular workload where:

$$MOD=\rho*MID \qquad \textbf{(1)}$$

Shuffle out data (SOD) is the same as the MOD, because

MOD is the source of shuffle. Different with the others, the shuffle in data (SID) in a node is not determined by map tasks but the proportion of local reducers' number and the whole reducers' number.
We use formula 2 to define the type of CPU-Bound workload. As for a map task, the operations of the I/O in disk include input, output, shuffle out and shuffle in. In the process of program running, every node has *n* map tasks which are synchronously running. Multiple tasks share the disk I/O bandwidth when the system stably runs. In our opinion, if the summation of MID+MOD+SOD+SID of *n* map tasks divided by MTCT is still smaller than the bandwidth of disk I/O, then this kind of task is CPU-bound. This is an upper-bound of
CPU-bound jobs. Formula 2 witnesses this conclusion, and it indicates the estimation of the program action.

$$\frac{(MID+MOD+SOD+SID)*n}{MTCT} = \frac{n*(1+2p)\,MID+SID}{MTCT} \geq DIOR \qquad \textbf{(2)}$$

The second class of workload is different with the former one. Its map tasks are CPU-Bound without shuffle action.
However when shuffle action begins, they block in I/O bandwidth. In this class, the ratio of the I/O data of map tasks to the runtime is less than DIOR. But when reduce phase begins, shuffle will generate lots of disk I/O, and it will make map task block in disk I/O. The CPU utilization ratio of this kind of job wouldn't reach 100%. We name this class of job as
Class Sway. This class means:

$$\frac{(MID+MOD+SID+SOD)*n}{MTCT} = \frac{n*((1+2p)MID+SID)}{MTCT} \geq DIOR \qquad \textbf{(3)}$$

Formula 4 define the type of I/O-Bound workload. In this class, every map task would generate lots of I/O operations in short time. And when *n* map tasks synchronously run on every node, there will be contention among different tasks. Even if reduce shuffle doesn't start, map task will be still bound to I/Formula 4 shows this relation. With our analyzing of single task, we could conclude that if *n* tasks of this kind synchronously run in the system, it will make application block at disk IO.

$$\frac{(MID+MOD)*n}{MTCT} = \frac{n*(1+p)\,MID}{MTCT} \geq DIOR \qquad \textbf{(4)}$$

We assume that every reducer have a similar size data input.
So the SID in MapReduce depends on the distribution of reducer in the cluster. The shuffle input data in every node relies on the proportion of running reducer number (RRN) to the whole reducer number (WRN).So the SID equals the following formula.

$$SID = \frac{RRN}{WRN} * SOR * \text{nodes number} \qquad \textbf{(5)}$$

The shuffle input data in every node relies on the proportion of running reducer number (RRN) to the whole reducer number (WRN).So the SID equals the following formula.

C.Hadoop Schedule Model:

Current Hadoop scheduler serves as a FCFS queue with priority. In the MapReduce framework, assigning happens when a TaskTracker which is in charge of the running jobs in the cluster heartbeats the JobTraker. The TaskTracker do heartbeat in every interval (default as 5 second) or when a task in that node finishes. Hadoop also uses a concept of slot number for each node to control maximal tasks running number of that node. The slot number actually depicts the maximal parallelizability of the machines. It can be configured in an xml file on every machine based on the hardware of that node. The current scheduler in Hadoop always assigns tasks from the job in the top of queue. It makes that map tasks from the same jobs are always running together. Because map tasks from the same job in MapReduce have similar behaviours, this kind of one queue scheduler could not efficiently use both CPU and I/O resource of the cluster. The contention among the similar tasks decreases the system throughput. In our work, we use three queues to separately execute different type of jobs.
Our experiment has shown that our approach can increase the system throughput in 30%.

D. Scheduling Strategies:

We perform simulations with the following three scheduling strategies along with our offline and online algorithms.

Shortest Job First (SJF). We assign tasks whose total processing time (*i.e.*, job size) is the shortest to a cluster first. This strategy appears to be a reasonable greedy algorithm minimizing the completion time, so we take it into account.

Shortest Task First (STF). The difference between SJF and this strategy is that STF only uses information that is locally available in determining the scheduling: the sizes of individual tasks submitted to a machine.

First-In First-Out (FIFO). Tasks are served in the order of arrivals without introducing any reordering of jobs. This strategy is our basis of comparison throughout this section

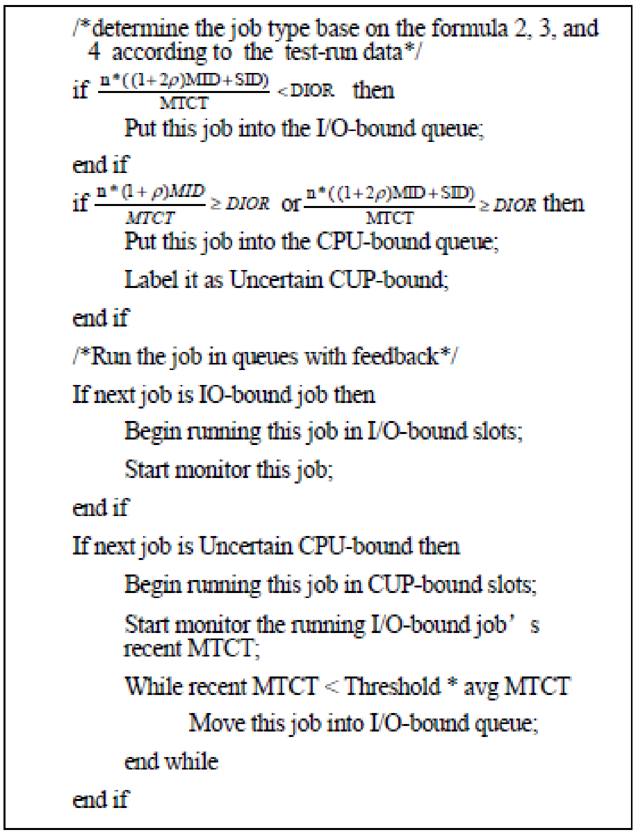
IV. TRIPLE QUEUE SCHEDULER

mmar. Current Hadoop scheduler implementation assigns tasks sequentially by using one queue. Other map tasks wouldn't be assigned until tasks from the job in the top of queue finish. This FCFS strategy works well when the jobs in the queue are of the

same class. However, I/O-bound tasks cause the CPUs. At the same time, the effect of the disk performance is on the opposite: I/O-bound tasks keep the disks busy, while CPU-bound tasks leave them idle. This phenomenon raises the problem of inadequately using of resource. This could happen in a real data centre where diverse kinds of workloads often run on it simultaneously.
The main idea of our work is that balancing different kinds of tasks could increase the utilization rate of both CPU and I/O bandwidth. The rationale for such paralleling is that these different tasks will hardly interfere each other's work, as they use different devices [7]. Therefore, they will work well eparately in the system. If the I/O operations' time is not negligible relative to the CPU time, such an overlap of the I/O activity with CPU work can be efficient [6].

M.R Predict

In this triple-queue scheduler, the predicting of workload type is essential. The characteristics of a task can be assessed by looking over its history. We assume that tasks from one job have similar characteristics. It means we can predict tasks' behaviour from the tasks already ran. We propose a new MapReduce workload predict mechanism called MR-Predict.
As shown in Figure 2, our scheduler determines a job's type based on the Formula 2, 3 and 4 which we proposed in previous chapter, and then jobs will run in two different queues with feedback. When a new job comes in, it will be put into the waiting queue first. Then the scheduler will assign one map task of that job to every Task Tracker when it has idle slots.
When these map tasks finish, we calculate the MTCT, MID and MOD by using the data form these tasks. Then jobs can be divided into three types. Each type can be determined by using these data by using our classification of MapReduce framework.
In this triple-queue scheduler, the predicting of workload type is essential. The characteristics of a task can be assessed by looking over its history. We assume that tasks from one job ave similar characteristics. It means we can predict tasks' behaviour from the tasks already ran. We propose a new MapReduce workload predict mechanism called MR-Predict.
As shown in Figure 2, our scheduler determines a job's type based on the Formula 2, 3 and 4 which we proposed in previous chapter, and then jobs will run in two different queues with feedback. When a new job comes in, it will be put into the waiting queue first. Then the scheduler will assign one map task of that job to every TaskTracker when it has idle slots. When these map tasks finish, we calculate the MTCT, MID and MOD by using the data form these tasks. Then jobs can be divided into three types. Each type can be determined by using these data by using our classification of MapReduce framework.
Among all the classifications we discussed above, IObound is one conservative class. Because every node may have existed jobs, their contention for resource would lead longer runtime of tasks. Consequently, certain kinds of tasks are likely to be classified as CPU-bound according to our formulas.
Therefore, our workload predict system includes one defect modify mechanism. After one task is assigned to the CPUBound queue, the system will monitor the running tasks in IOBound queue, if the MTCT of recent tasks increase to a certain threshold, which we define as 140% in our system, then we get the conclusion that the task which we just assigned shouldn't be allocated to the CPU queue, we need to re-allocate it to the I/O-Bound queue.



```
/*determine the job type base on the formula 2, 3, and
  4 according to the test-run data*/
if $\frac{n*((1+2\rho)MID+SID)}{MTCT} < DIOR$ then
    Put this job into the I/O-bound queue;
end if
if $\frac{n*(1+\rho)MID}{MTCT} \geq DIOR$ or $\frac{n*((1+2\rho)MID+SID)}{MTCT} \geq DIOR$ then
    Put this job into the CPU-bound queue;
    Label it as Uncertain CUP-bound;
end if
/*Run the job in queues with feedback*/
If next job is IO-bound job then
    Begin running this job in I/O-bound slots;
    Start monitor this job;
end if
If next job is Uncertain CPU-bound then
    Begin running this job in CUP-bound slots;
    Start monitor the running I/O-bound job' s
    recent MTCT;
    While recent MTCT < Threshold * avg MTCT
        Move this job into I/O-bound queue;
    end while
end if
```

Pseudo-code of the schedule policy

B.Schedule Policy

The triple-queue tasks scheduler contains a CPU-bound queue where jobs of CPU-Bound Class stand in, an I/O-bound queue where jobs of I/O-bound Class stand in, and a waiting queue where all jobs stand in before their type is determined.
When a new job comes in, it will be added to the waiting queue. Then the scheduler assigns one map tasks to every
Task Tracker for predicting the job type. As shown in the figure 3, if both the CPU-bound queue and I/O-bound queue are empty at that time, the job on the top of waiting queue would move to the idle queue and keep running until its type is determined. And then, as shown in Figure 4, if the undetermined job is found stand in a wrong queue, it will move to the right one.
CPU-Bound queue and I/O-bound queue have their own map slot number and reduce slot number which can be configured by users based on the information of cluster hardware. Each queue works independently, and serves a FCFS with priority strategy just like Hadoop's current job queue. On every node, both of the queues have their own slots for running certain kind of jobs unless one queue becomes empty. In this situation, idle slots will be fully used by the running job until a new job adds in the empty queue.

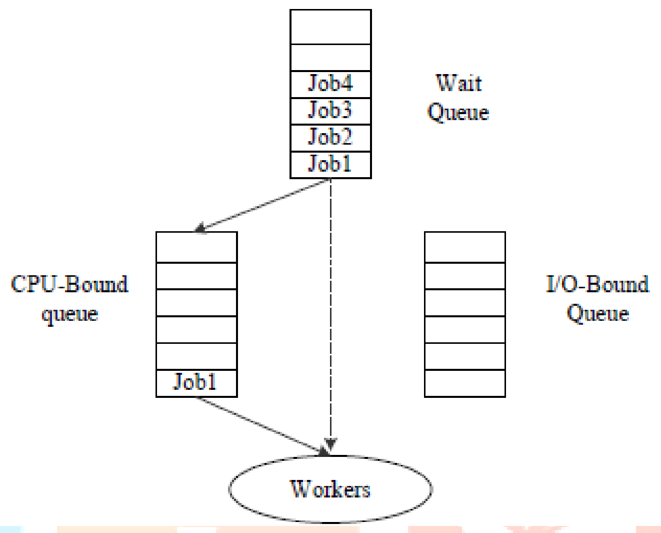


Figure 3.The Schedule Policy when both queues are empty

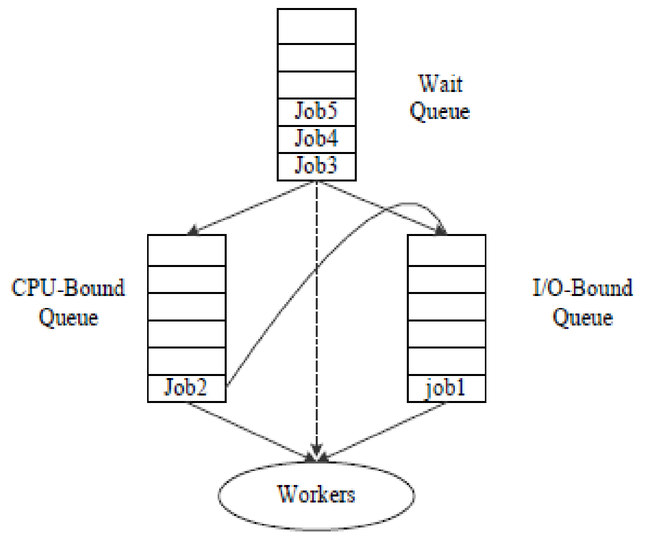


Figure 4. Jobs which are detected standing in a wrong queue will switch to another queue

IV. EVALUATION

We start our evaluation by compiling statistic of jobs to verify our workload classification. Then we do a couple of experiments to validate that one queue schedule cannot raise the utilization of both I/O and CPU. At last we run a suit of mixed type jobs for validating the triple queue scheduler works well in a multiple workloads environment.
We use a local cluster to test our triple queue scheduler which contains 6 DELL1950 nodes connected by gigabit
Ethernet. Each node has 2 Quad Core 2.0GHz CPU, 4GB memory, 2 SATA disks..

A.Resource Utilization

We run a couple of jobs to evaluate the hardware utilization of the current one queue scheduler. We choose three jobs which belong to the three different workload types [2] Says that short jobs are a major use of MapReduce. For example, Yahoo won the TeraSort 1TB benchmark using 910 nodes in 209 second [8], and the average MapReduce job at
Google in September 2007 was 395 seconds long [1]. So we choose the input data set as 15GB for each job which could simulate the situation in a real product environment. In these experiments, we set the map slots and the reduce slot to 8. So the n of formula 2, 3 and 4 is 8.
We use a testing program called DIO to get the DIOR value of ideal performing Hadoop system. This program runs without reduce phase, and its programming act is simply to read and write on the disk. Therefore, it is totally IO intensive.
We could estimate the system's DIOR value according to the average runtime of single task in this program. In our experiment, we showed that our DIOR is almost 31.2 MB/S.\

Program	**MID**	**MOD**	**MTCT**
Terasort	64M	64M	8sec
Grep-Count	64M	1M	91sec
Word-Count	64M	64M	35sec
DIO	64M	64M	4.1sec

Table 1.Test jobs

As shown in table 1, the first job is TeraSort [8] which is a famous total order sort benchmark. TeraSort is essentially a sequential I/O benchmark. It's MID is 64M, MOD is almost 64M in average. The MTCT of this program is 8 second.
According to our formula 4, this job belongs to I/O-bound class.
The second job is Grep-Count which based on the commonly used program Grep in Linux. The Grep-Count program accepts a regular expression and the files as the input parameters. Unlike Grep in Linux, this Grep-Count output the occupation number of the same lines matching the input regular expression in the documents rather than output all lines matching the regular expression. The computing complexity depends on the input regular expression. In our test case, we use [.]* as the regular expression which make the job be CPUBound. Its MID is 64M and MOD is almost 1M in average. According to our formula 2, this job belongs to I/O-bound class.
The third job is WordCount. It splits the input text into words, shuffles every word in map phase and counts its occupation number in reduce phase. Its MID is 64M and MOD is almost 64M in average. According to our formula 3, this job belongs to sway class. Among the testing results, the CPU utilization rate of TeraSort always keeps in low level. That's because the task is
IO-bound, and the CPU utilization rate couldn't rise. On the Contrary, in the tests to Grep-Count, the CPU utilization rate is always nearly 100%. This also verifies that this task is of the class of CPU intensive. The performance of WordCount differs from the two programs above. When the program starts, because reduce task doesn't begin, the CPU utilization rate reaches 80%; but after reduce task begins, the CPU utilization rate rapidly decreases. [2] Says that short jobs are a major use of MapReduce. For example, Yahoo won the TeraSort 1TB benchmark using 910 nodes in 209 second [8], and the average MapReduce job at
Google in September 2007 was 395 seconds long [1]. So we choose the input data set as 15GB for each job which could simulate the situation in a real product environment. In these experiments, we set the map slots and the reduce slot to 8. So the n of formula 2, 3 and 4 is 8. We use a testing program called DIO to get the DIOR value of ideal performing Hadoop system. This program runs without reduce phase, and its programming act is simply to read and write on the disk. Therefore, it is totally IO intensive.
We could estimate the system's DIOR value according to the average runtime of single task in this program. In our experiment, we showed that our DIOR is almost 31.2 MB/S.\

Program	**MID**	**MOD**	**MTCT**
Terasort	64M	64M	8sec
Grep-Count	64M	1M	91sec
Word-Count	64M	64M	35sec
DIO	64M	64M	4.1sec

Table 1.Test jobs

As shown in table 1, the first job is TeraSort [8] which is a famous total order sort benchmark. TeraSort is essentially a sequential I/O benchmark. It's MID is 64M, MOD is almost 64M in average. The MTCT of this program is 8 second.

According to our formula 4, this job belongs to I/O-bound class. The second job is Grep-Count which based on the commonly used program Grep in Linux. The Grep-Count program accepts a regular expression and the files as the input parameters. Unlike Grep in Linux, this Grep-Count output the occupation number of the same lines matching the input regular expression in the documents rather than output all lines matching the regular expression. The computing complexity depends on the input regular expression. In our test case, we use [.]* as the regular expression which make the job be CPUBound. Its MID is 64M and MOD is almost 1M in average. According to our formula 2, this job belongs to I/O-bound class.
The third job is WordCount. It splits the input text into words, shuffles every word in map phase and counts its occupation number in reduce phase. Its MID is 64M and MOD is almost 64M in average. According to our formula 3, this job belongs to sway class. Among the testing results, the CPU utilization rate of TeraSort always keeps in low level. That's because the task is IO-bound, and the CPU utilization rate couldn't rise. On the Contrary, in the tests to Grep-Count, the CPU utilization rate is always nearly 100%. This also verifies that this task is of the class of CPU intensive. The performance of WordCount differs from the two programs above. When the program starts, because reduce task doesn't begin, the CPU utilization rate reaches 80%; but after reduce task begins, the CPU utilization rate rapidly decreases.

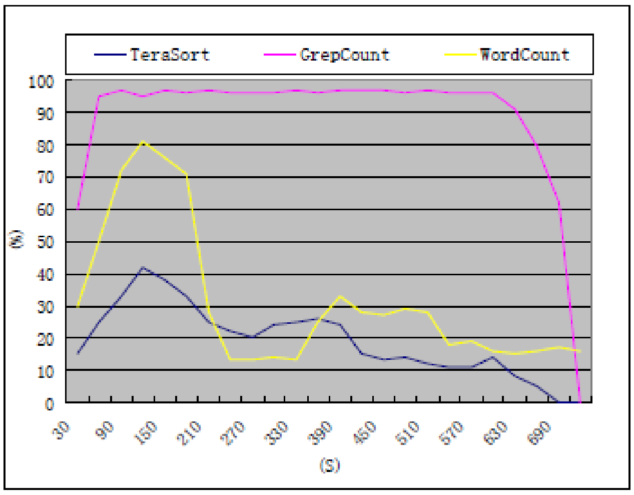


Figure 5. The average CPU utilization rate of the cluster

B.Triple Queue Scheduler experiment

In the previous section, our experiments have shown that current Hadoop scheduler indeed could not efficiently use both of the CPU and I/O resource. In this section, we design a new scene that is multiple jobs run together. We define three clients which submit the three different kinds of jobs in their separate sessions. One client submits one kind of job, and sequentially submits the same job after the previous one finishes. This scène is used for simulating the real product environment which periodically does the same kinds of job every day.
We choose the task used in last chapter as testing task. We use three different clients to constantly request for running these three kinds of tasks. Every job runs five times, and in total 15 jobs will run. The three jobs are of different type.
According to the throughput and complete time of these tasks, we could analyse the improvement of performance because of paralleling IO bound and CPU bound tasks.
In our experiments, the sequence of task executing would influence the result of the test. The time of Reduce-compute phase in TeraSort and WordCount is a little longer, while that in Grep-count is a little shorter. TeraSort under Reduce compute phase is CPU bound, and wordCount under Reduce compute phase is IO bound. In the scheduler of Hadoop, after the Map phase of TeraSort finishes, map slot will be idle, if the next job in the queue is Grep-count, the map tasks of Grepcount and TeraSort could be parallel. It makes the integrity of the system improve in certain level. Therefore, in this test, we give the data under the best, the worst and the average condition.

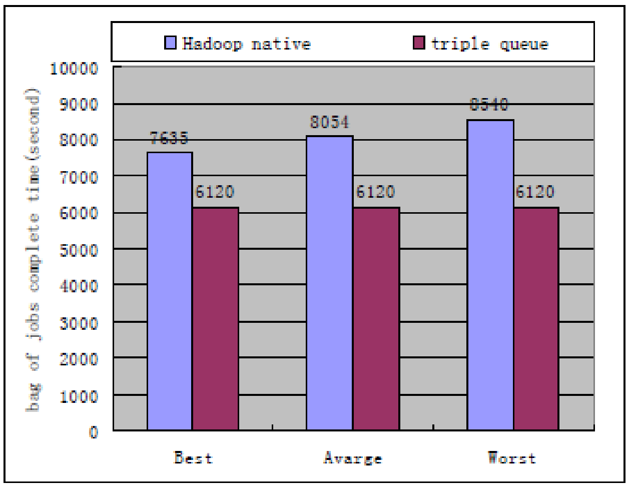


Figure 6. Makespan of the Triple-Queue scheduler against Hadoop native scheduler

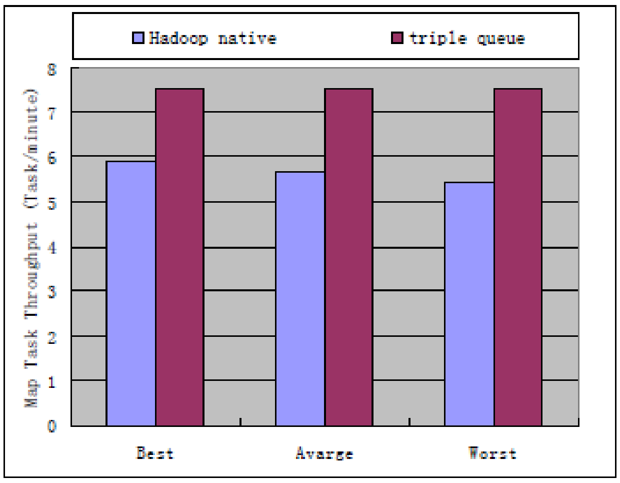


Figure 7. The throughput of map tasks

As shown in figure 5 and 6, the testing data has witnessed that with the scheduler of Hadoop, by adjusting the sequence of jobs, the performance of system is different. The make span [18] of Hadoop native scheduler is about 7635 seconds in the best condition as well as 8540 seconds in the worst condition. The throughput of map tasks is 5.89 in the best condition and 5.44 in the worst condition. The triple queue scheduler can significantly improve the system performance. It improves the throughput by 30% in map-shuffle phase, and enhances the make span by 20% under parallel workloads

RELATED WORK

The scheduling of a set of tasks in a parallel system has been investigated by many researchers. Many schedule algorithms has been proposed [11, 12, 13, 16, 17]. [16, 17] focus on scheduling tasks on heterogeneous hardware, and [11, 12, 13] focus on the system performance under diverse workload. The heterogeneity of workloads is also in our assumptions.
In fact, it is nontrivial to balance the use of the resources in applications that have different workloads such as large computation and I/O requirements [10]. [6, 14] discussed the problem of how I/O-bound jobs affect system performance, and [7] shown a gang schedule algorithm which parallel the CPUbound jobs and IO-bound jobs to increasing the utilization of hardware. Our work shares the some ideas with these work
However, we depict the different kinds of workloads in the
MapReduce system.
The schedule problem in MapReduce also attracted many attentions. [2] Addressed the problem of how to robustly perform speculative execution mechanism under heterogeneous hardware environment. [9] Derive a new family of scheduling policies specially targeted to sharable workloads.
Hadoop is a popular open source implementation of
MapReduce [1] and Google File System [15]. Yahoo and
Facebook also designed schedulers of Hadoop as Capacity scheduler [4] and Fair scheduler [5]. These two schedulers used multiple queues for allocating different resources in the cluster.
They both provide short response times to small jobs in a shared Hadoop cluster. Our work has something in common with these two schedulers. However, our work focuses on the utilization of hardware under heterogeneous workloads. We give an automatic

workload predicting mechanism for detecting the workload type on the fly. Then the triple-queue scheduler use two different queues for assign tasks of different types.

V. CONCLUSION AND WORK LOAD

This paper discusses the MapReduce performance under heterogeneous workloads. We analyse the typical MapReduce workloads on the MapReduce system, and classify them into. three categories. We propose the Triple-Queue scheduler based on the classification. The triple-queue scheduler dynamically determines the category of one job. It contains a waiting queue
to test-run new joined job and to predict the workload type of this job according to the result. It also includes a CPU-bound queue and an I/O-bound queue for paralleling different kinds of jobs. According to the experiment results, the scheduler can correctly distributes jobs into different queues in most situations. And then the job will run due to the resource appointed by the queue. Our experiments have shown that the Triple Queue Scheduler could increase the map tasks' throughput of the system by 30%, and save the make span by 20%. In our work, we assume that the distribution of workload is uniform. We then predict the future behaviours of tasks base on these distribution. In our future work, we will try to predict the workloads which are of different kinds of distribution and consider the hardware heterogeneity of the Hadoop cluster environment.

REFERENCES

[1] Andrea C. Arpaci-Dusseau, Remzi H. Arpaci-Dusseau, David E. Culler, Joseph M. Hellerstein, and David A. Patterson. High-performance sorting on networks of workstations. In *Proceedings of the 1997 ACM SIGMOD International Conference on Management of Data*, Tucson, Arizona, May 1997.

[2] Remzi H. Arpaci-Dusseau, Eric Anderson, Noah Treuhaft, David E. Culler, Joseph M. Hellerstein, David Patterson, and Kathy Yelick. Cluster I/O with River: Making the fast case common. In *Proceedings of the Sixth Workshop on Input/Output in Parallel and DistributedSystems (IOPADS '99)*, pages 10.22, Atlanta, Georgia, May 1999.

[3] Arash Baratloo, Mehmet Karaul, Zvi Kedem, and Peter Wyckoff. Charlotte: Metacomputing on the web. In *Proceedings of the 9th International Conference on Parallel and Distributed Computing Systems*, 1996.

[4]] Luiz A. Barroso, Jeffrey Dean, and Urs H¨olzle. Web search for a planet: The Google cluster architecture. *IEEE*
Micro, 23(2):22.28, April 2003.

[5] John Bent, Douglas Thain, Andrea C.Arpaci-Dusseau, Remzi H. Arpaci-Dusseau, and Miron Livny. Explicit control in a batch-aware distributed _le system. *Proceedings of the 1st USENIX Symposium on Networked Systems Design and Implementation NSDI*, March 2004.

[6]] Guy E. Blelloch. Scans as primitive parallel operations. *IEEE Transactions on Computers*, C-38(11), November 1989.

[7]] Armando Fox, Steven D. Gribble, Yatin Chawathe, Eric A. Brewer, and Paul Gauthier. Cluster-based scalable network services. In *Proceedings of the 16th ACM Symposium on Operating System Principles*, pages 78. 91, Saint-Malo, France, 1997.

[8] Sanjay Ghemawat, Howard Gobioff, and Shun-Tak Leung. The Google _le system. In *19th Symposium on O Operating Systems Principles*, pages 29.43, Lake George, New York, 2003.