Dynamic Split Model of Resource Utilization in MapReduce

Xiaowei Wang
Institute Of Computing Technology
Chinese Academy Of Sciences
jessica_2011_gucas@-126.com

Jie Zhang
Institute Of Computing Technology
Chinese Academy Of Sciences
zhangjie@-software.ict.ac.cn

Huaming Liao
Institute Of Computing Technology
Chinese Academy Of Sciences
lhm@ict.ac.cn

Li Zha
Institute Of Computing Technology
Chinese Academy Of Sciences
char@ict.ac.cn

ABSTRACT

MapReduce is gaining increasing popularity as a parallel programming model for large-scale data processing. We find however some traditional MapReduce platforms have a poor performance in terms of cluster resource utilization since the traditional multi-phase parallel model and some existing schedule policies used in the cluster environment have some drawbacks. We address these problems through our experience in designing a Dynamic Split Model of the resources utilization which contains two technologies, Dynamic Resource Allocation considering the phase priority as well as job requirement when allocating resources and Resource Usage Pipeline which can assign tasks dynamically. We verify our optimization on top of Hadoop and the results show that these technologies can improve the throughput by 21.72%, the average wall time gain is 55.83%. And we improve the percentage of user CPU utilization by 12.93%, reduce the percentage of iowait CPU and idle CPU utilization by 6.61% and 6.73%. The upstream speed and downstream speed are increased by 11.3% and 23.5%. What's more, we have relieved the Disk I/O bottleneck by 30.3%.

Categories and Subject Descriptors

C.2.4 [**Computer Systems Organization**]: Distributed Systems

General Terms

Algorithms, Design, Performance

Keywords

MapReduce, Dynamic Schedule, Resource, Pipeline, Parallel Process

DataCloud-SC'11, November 14, 2011, Seattle, Washington, USA.

1. INTRODUCTION

We are entering a new era of data deluge with technologies to store and process massive data becoming more widespread. So the system which can satisfy the above meets takes on an urgency, which we refer to as Data-Intensive Super Computing (DISC) system [2]. MapReduce [4] is a programming model put forward by Google used for large-scale data processing. Some DISC system platforms are built on top of MapReduce: Dryad [7] introduced by Microsoft uses a Directed Acyclic Graph (DAG) based execution model in its distributed execute engine, and Hadoop [1] is the open source version of Google MapReduce. The mainstream of parallel processing also captures the academic world attention. Sector/Sphere [5] is a distributed computing platform similar to Google GFS/MapReduce. It contains a parallel runtime Sphere as well as a distributed file system Sector. Another parallel runtime is phaser [9] which is a coordinate construct for dynamic parallelism under the environment of multi-processors instead of multi-nodes.

The essence underlying these parallel programming frameworks mentioned above is that they are all derived from multi-phase parallel model [6]. The traditional multi-phase model has bad performance on the resource utilization efficiency. This problem comes from two aspects. On the one hand different phases have different priorities as well as different resource usage bias and must be executed strictly to that priority which causes resource usage unbalance. Some methods can loosen the strict execution order among different phases by sub-operations overlap execution such as phaser accumulator [8] or streaming pipeline in Hadoop Online prototype [3]. However the unbalance still exists if the resource need in sub-operations' overlap execution is the same. On the other hand some DISC platforms ignore the cluster load and the jobs requirement when they allocate resource. Take Hadoop for example, it allocated resource by a static configuration and can not be changed according to the cluster load. This can lead to a "slot hoarding" problem and a "resource allocation unbalance" problem which could cause inefficient resource usage.

Our technology in this paper to address the above problem is called Dynamic Split Model of Resources Utilization which

includes two basic technologies: Resource Usage Pipeline (RUP) and Dynamic Resource Allocation (DRA). In order to make the resource pipeline feasible we need to take some measures to divide the mixture of different resource usages into separate ones dynamically and launch a task at a proper point. Another technology Dynamic Resource Allocation will count the system load and the status of each job in order to allocate our resource more efficient.. And we use Hadoop to verify that our optimization. We have implemented and tested the technologies RUP and DRA. And the final experiments show that our overall optimization can offer a performance gain over the conventional one by about 21.72%, improve the average will time by 55.83%, and use resource more reasonable and efficient. Although our optimization is implemented on Hadoop, the problem we address is by no means constrained to Hadoop even MapReduce computing model. In general effective resource utilization can provide a more significant performance in DISC system.

The rest of paper is organized as follows. Section 2 provides background of Hadoop and the resource utilization problem as well as our solution is described in Section 3. Section 4 presents our Dynamic Resource Allocation technology, and the Resource Usage Pipeline is provided in Section 5. Our optimization is evaluated in section 6. We will surveys some related work in section 7 and conclude in section 8.

2. BACKGROUND

MapReduce is a programming model which is well-suited in DISC system. A job expressed in the form of MapReduce usually consists of two sequential phases Map and Reduce. In the Map phase, users can apply their user-defined map function to input data which is a set of (key, value) pairs. The intermediate output of another kind of (key, value) pairs is produced and transferred to reduce function. Reduce function will process all values belong to the same key one time and output the final (key, value) pairs.

Hadoop is the most popular open source implementation of Google GFS/MapReduce. It is made up of two parts Hadoop Distributed File System (HDFS) and MapReduce compute framework. MapReduce framework is built on top of HDFS containing a Job Tracker and several Task Trackers. A job submitted to the Job Tracker is made up of two phases map phase and reduce phase. In the map phase the job is divided into map tasks whose number is equal to the amount of split and in the reduce phase reduce tasks number of a job is decided by the configuration.

2.1 Map Task

A map task belonging to a specific job will read a split which is a portion of the input file from HDFS. The execution steps are listed as follows.

- MapRunner reads a (key, value) pair from the assigned split in HDFS using RecordReader and applies the user-defined map function to the pair.
- After processing one pair, map task outputs the result to a fixed-size buffer. If the buffer is overflow it will apply a quick-sort to the full buffer first and then store the content of the buffer to a spill file so the buffer becomes empty and can therefore receive more results. This process will take place repeatedly until all records are used up.
- Map task merges all the spills on the disk into a sort file and stores it to the local file system as well as a corresponding index file referring to that data file.

Task Tracker will ask for a new task from Job Tracker as soon as this map task finishes and releases a slot.

2.2 Reduce Task

A reduce task will not start until a portion of map tasks belonging to the same job finish and commit their output to local disk. A reduce task execution is divided into three phases.

- In the shuffle phase reduce task fetches a specific partition of every map task output using HTTP request and puts them into memory first. If the memory is full it will apply merge sort to the memory and output the result to a temp file.
- After all partition is fetched, reduce task enters the sort phase where it sorts all sorted files stored in memory or disk using merge sort grouping all records to the same key together.
- The reduce phase applies the user-defined reduce function to each key and the set of values belonging to the same key.

3. DYNAMIC SCHEDULING MODEL

Figure 1 shows the real execution situation on a single node in the raw version hadoop. Where we can see the raw version hadoop does have several flaws which puts a serious impact on the system resource efficient utilization.

- Single node resource usage unbalance: Figure 1 tells us within a round of map task or reduce task different phases have different resource usage bias especially when homogenous tasks execute at the same pace some resources such IO or CPU may be underused while the others overused.
- Reduce slot hoarding: In MapReduce, each reduce copies its portion of the results of each map, and can only apply the user's reduce function once it has results from all map. However if we submit a big job, which will take a long time to finish all map tasks. It will hold reduce slots for a long time until all the map tasks are finished, and starve small jobs and underutilize resource.
- Resource allocation unbalance within job: MapReduce will allocate resource by a static configuration, which does not consider the system load and the jobs requirement. This will lead to resource allocation unbalance. From Figure 1 we can see no matter how many map tasks and reduce tasks to run, the system slot number is never changed.

By and large there is a mismatch between the dynamic workload and static resource assignment. To address the above problems we develop a Dynamic Split Model of Resource Utilization to make a more reasonable and intelligent use of a single node resource . Figure 2 demonstrates our design result.

- We separate resource usage within a phase into two periods: CPU period and IO period. And we use advanced scheduling to launch a task at a proper point. It is shown in Figure 2 where one task's sub-operation can overlapped with the other in case their resource usage is complementary.

- We collect the system load and the status of each job at run time to allocate resource dynamically. Figure 2 illustrates this clearly that at the three arbitrary time point the number of slot is not the same and can be modified according to system load.

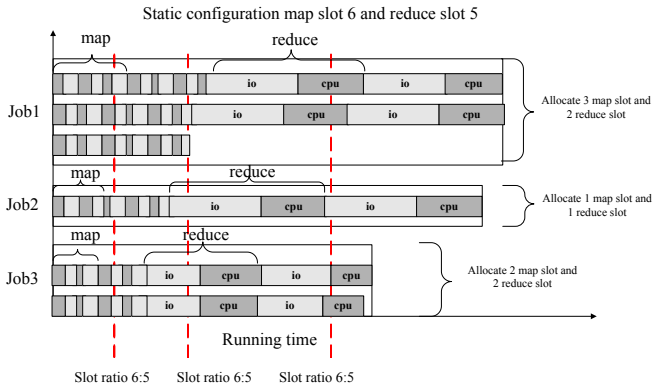

Figure 1: job execution situation on a single node in raw version hadoop the red dash dotted line stands for three arbitrary time point in the execution process.

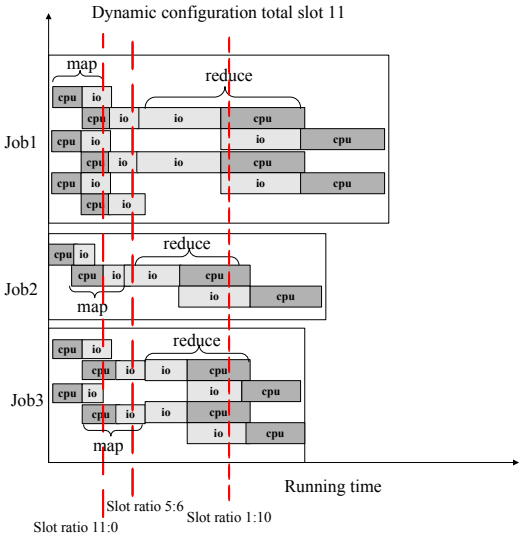

Figure 2: job execution situation on a single node in new version hadoop the red dash dotted line stands for three arbitrary time point in the execution process

Figure 2 illustrates this clearly that at the three arbitrary time point the number of slot is not the same and can be modified according to system load.

4. DYNAMIC RESOURCE ALLOCATION

4.1 Reduce Slot Hoarding Problem

MapReduce normally launches reduce tasks for a job as soon as its first few maps finish, so these reduces can begin copying map outputs while the remaining maps are running. However, in a large job, the map phase may take a long time to complete. The job will hold any reduce slots it receives during this until its maps finish. So the other jobs, which submitted later, will starve until the large job finishes. This is called "reduce slot hoarding" problem, which will waste resource and delay the wall time of jobs.

It may appear that this problem could be solved by starting reduce tasks later or making them suspended, but [10] pointed out that this solution is not feasible. And their solution is to split reduce tasks into two logically distinct types of tasks, copy tasks and compute tasks, with separate forms of admission control. This solution can relieve this problem, but powerless for the "resource allocation unbalance" problem, which will be described in the next section.

4.2 Resources Allocation unbalance Problem

Another issue we meet in MapReduce is that the MapReduce will allocate the maximum number of slots by a static configuration, such as the parameters: mapred.tasktracker.map.tasks.maximum and mapred.tasktracker.reduce.tasks.maximum, and will never be changed when the cluster is running. However,the requirement for slots varies along with job proceeding. For example, if a job is submitted at its first several seconds, it just needs map slots, because there is not data for reduce tasks to copy; Then, if some map tasks are finished, it needs more map slots to run map tasks and a little reduce slots to copy output; when all map tasks of the job are finished, it just need reduce slots to copy output and apply the user's reduce function. Obviously, static slot configuration can't adapt to these requirements. It will cause a problem that, sometimes the cluster has a surplus map slots but reduce slots insufficient, and vice versa. This is called "resource allocation unbalance" problem, which will lead to poor performance.

4.3 Our Solution: Dynamic Resource Allocation

Our proposed solution to these problems is dynamic resource allocation. We will allocate resource according to the cluster load and all jobs run-time status.

As we mentioned above, during the execution of a job, the resource requirement is changing every time. We need more map slots at the beginning; with the map tasks' completion, the required map slots is gradually decreased, but the requirement to reduce slots increased; And more reduce slots are needed when all map tasks are finished. Therefore, for a job, the resource requirement is changing with job status changing. Here the job status means the completion rates of map tasks and reduce tasks. So we define the dynamic weight of the map phase and the reduce phase according to the job status to simulate the dynamic requirement for resource.

Suppose the weight of map phase is w_m, the weight of reduce phase is w_r. We normalized the weight:

$$w_m + w_r = 1 \tag{1}$$

Suppose: the percentage of map phase completion is x.

$$x = F_{task}/T_{task} (0 <= x <= 1) \quad (2)$$

Where F_{task} is the number of finished map tasks, T_{task} is the total number of map tasks in the job. And We hope the variation trend of weight with the changing of job status, as shows in figure 3. We have tried straight-line formula, polynomial formula and exponential formula to simulate the dynamic weight, and found the exponential formula closest to the curves shows in figure 3. So we defined the w_m as bellow:

$$w_m = 1 - \frac{1}{e^6 - 1}\left(e^{6x} - 1\right) \quad (3)$$

Because of Equation(1), we have the following:

$$w_r = 1 - w_m = \frac{1}{e^6 - 1}\left(e^{6x} - 1\right) \quad (4)$$

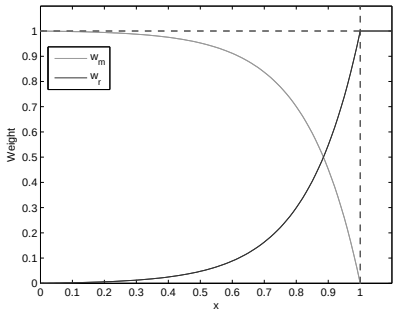

Figure 3: The dynamic weight of map phase and reduce phase with the changing of job status.

If only one job in the cluster, then the job could take up resource alone. So we can allocate resource according to w_m and w_r.

Suppose the number of slots in the cluster is R, and R_m slots use for map phase, R_r slots use for reduce phase. Then we get:

$$R_m = \frac{w_m R}{w_m + w_r} = w_m R = (1 - \frac{e^{6x} - 1}{e^6 - 1})R \quad (5)$$

$$R_r = \frac{w_r R}{w_m + w_r} = w_r R = \frac{(e^{6x} - 1)}{e^6 - 1}R \quad (6)$$

So there are R_m map slots, and R_r reduce slots in cluster, and if there is a map task finished, the cluster will automatically re-allocate the resource.

In a cluster, there may be lots of users submit jobs, one user may submit some jobs at the same time, and jobs submitted by different users may have different weights. To describe this scenario, we suppose: there are n jobs running in the cluster, and each job i has a weight w_i. The resource for job i is R_i, and the map phase gets resource R_{im}, the reduce phase gets resource R_{ir}.

Definition: in a cluster, a job will get the resource by its weight w_i, and within the job, map phase and reduce phase will get resource by w_m and w_r. So job i gets resource:

$$R_i = \frac{w_i}{\sum_{i=1}^{n} w_i} R \quad (7)$$

Resource allocated for the map phase and reduce phase of this job is:

$$R_{im} = \frac{w_{im} R_i}{w_{im} + w_{ir}} = w_{im} R_i = (1 - \frac{e^{6x_i} - 1}{e^6 - 1}) \frac{w_i}{\sum_{j=1}^{n} w_j} R \quad (8)$$

$$R_{ir} = \frac{w_{ir} R_i}{w_{im} + w_{ir}} = w_{ir} R_i = (\frac{e^{6x_i} - 1}{e^6 - 1}) \frac{w_i}{\sum_{j=1}^{n} w_j} R \quad (9)$$

We have got the resources allocation for map phase and reduce phase of each job, so we can calculate the total resources allocation for map slots and reduce slots. Suppose there are totally R_M resources allocated for map slots, and R_R resources allocated for reduce slots. Then:

$$R_M = \sum_{i=1}^{n} R_{im} = \sum_{i=1}^{n} (1 - \frac{e^{6x_i} - 1}{e^6 - 1}) \frac{w_i}{\sum_{j=1}^{n} w_j} R \quad (10)$$

$$R_R = \sum_{i=1}^{n} R_{ir} = \sum_{i=1}^{n} (\frac{e^{6x_i} - 1}{e^6 - 1}) \frac{w_i}{\sum_{j=1}^{n} w_j} R \quad (11)$$

The R_M and R_R define how many resources used for map slots and reduce slots. When a job is finished, a map task is completed, or a new job is submitted, all resource allocation formulas will be recalculated automatically, then we can get a new allocation of resources. Therefore the dynamic resource allocation will allocate resource dynamically according to all jobs status. If we submit a big job in the cluster, the map phase of this job will get more resources, and the reduce phase will get little until the map phase is close to finish. So we solve the "reduce slot hoarding" problem using dynamic resource allocation. When the cluster has a free slot, the dynamic resource allocation will first determine whether this slot is used for the map slot, or reduce slot; Then ranks all jobs in descending order according to w_m or w_r; Finally, allocates the map slot or reduce slot to the first job.

5. RESOURCE USAGE PIPELINE

5.1 Resource Usage Unbalance problem

There is a configuration contradiction in the parameter io.sort.mb, a size of buffer to hold map output. If we set this value too large most map tasks could stay in the same period IO or CPU for a longer time which results in one kind of resource usage conflict while the other idle. However too small will lead to IO overhead or more frequent disk IO. The probable value is difficult to set due to the variable output volume. As for the reduce phase it has a obvious resource usage bias so if the reduce tasks proceeds at almost the same pace it could also cause the resource usage unbalance problem in a single node.

5.2 Our Solution: Resource Usage Pipeline

To address the single node resource usage unbalance problem within a phase, we develop a technology called Resource Utilization Pipeline (RUP). On the contrary to the traditional task parallelism it provides a finer granularity parallel mechanism to maximize the complementation of different resources usage. That means not only can different tasks run

simultaneously but also different resource usage periods of different tasks can be overlapped with each other. In map phase we use dynamic enlargeable buffer instead of fixed-size buffer to hold output results which avoids the intermediate temporary file IO under most situations and separates the mixture resources usage period into two individual ones: CPU and IO. And we use dynamic slot request to schedule tasks in advance to guarantee the above two resource utilization period can be overlapped with each other.

5.2.1 Dynamic Buffer Enlargement in Map Phase

There are two kinds of buffer in map phase: buffer to hold (key, value) pairs, kvbuffer for short and buffer containing indexes referring to the location of (key, value) pairs in kvbuffer which we call kvoffsets. Both buffer size is decided by io.sort.mb factor. The logic of dynamic buffer enlargement applies to both of them. Therefore we use kvbuffer as an example to demonstrate our technology. The following procedure is the dynamic buffer enlargement logic:

- Map task allocates a kvbuffer according to the default io.sort.mb value to hold output (key, value) pairs.
- As the map function puts a (key, value) pair to kvbuffer, map task will judge whether kvbuffer is full. If kvbuffer is full it will calculate whether there is enough free JVM memory to allocate more buffer to hold more pairs. If there is we will firstly enlarge kvbuffer and then put the pair into it otherwise we will set a flag to indicate map task to handle the remaining output in the way the raw version hadoop does. The method we use to calculate how much free memory is needed is direct which involves four parameters: total file size (tfs), processed file size (pfs), and allocated buffer size (abs) and available free JVM memory (afm). The judgment formula is:

$$\text{pfs} : \text{tfs} = \text{abs} : \text{x} \tag{12}$$

  x here stands for the memory we need to allocate. If x is smaller than afm times a threshold kvbuffer can be expanded to x.

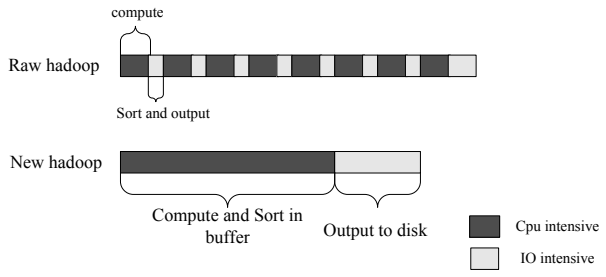

Figure 4: a map task process analysis from resource usage perspective.

The map phase can be divided into two periods through dynamic buffer technique each of which has an obvious resource usage bias. The first period is to apply map function to each input pair and output them to memory so it is CPU-intensive. The final results of a map task will be outputted to the local disk in the second period at a time which is IO-intensive. Figure 4 illustrates the traditional and optimized map phase process logic from the resource usage perspective.

5.2.2 Dynamic Slot Request of Map Task

Dynamic slot request is built on top of the dynamic buffer. If we merely use dynamic buffer it will cause resource conflict or idle described above. Because IO-intensive period and CPU-intensive period takes a longer time which leads to a higher probability that most map tasks stay in the same period. Dynamic slot request allows us to implement resource utilization pipeline within a map phase so as to different map task may stay different periods.To implement resource usage pipeline a map task needs to produce a new map slot which we call additional map slot at the point it enters into the IO-intensive period then the task tracker can ask for a new map task which we call additional map task different from the normal map task. The additional map task will enter the CPU-intensive period while the normal one enters IO-intensive period at the same time. Figure 5 illustrates the pipeline version compared to the raw version Hadoop. Dynamic slot request has a limitation that is only the map task whose output can be stored in the kvbuffer at a time is allowed to produce a new additional map slot. The additional map slot number is decided by configuration parameter mapred.tasktracker.map.tasks.maximum which we know is used for deciding how many map tasks is allowed to run at a single node simultaneously in the raw version Hadoop. In new version Hadoop it is interpreted into the sum of normal map slot and additional map slot, half of which is assigned to normal map task and the remaining is to additional map task.

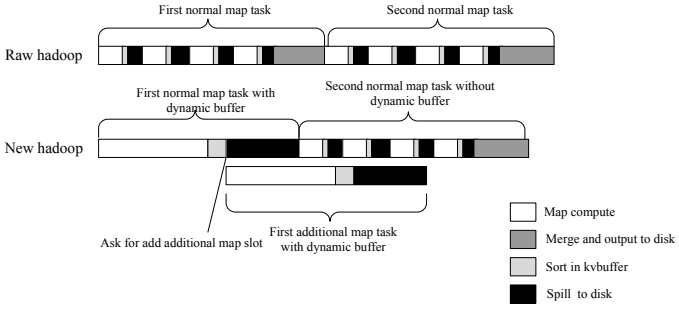

Figure 5: dynamic slot request

5.3 Dynamic Slot Request of Reduce Task

As it is mentioned in section 2 reduce phase has two distinct resource requirements periods: shuffle period and sort-compute period which provides the possibility to implement the resource utilization pipeline. What the new version does is analogy to map phase that is to assign a new reduce task to task tracker when a reduce task enters into sort-compute period so that the IO period of the latter task can be overlapped with the CPU period of the former task. However we failed to take it into consideration that system bottleneck varies because CPU and Net Work process power is unequal. We address this issue by dividing the configuration parameter mapred.tasktracker.reduce.tasks.maximum into three sub-parameters: mapred.tasktracker.reduce.toal.tasks, mapred.-tasktracker.reduce.shuffle.tasks and mapred.tasktracker.reduce.-compute.tasks which stand for separately the maximum reduce tasks number allowed running simultaneously at one node, the maximum number shuffling map output at a time and the maximum doing sort-computing at a time. We

place the following three admission constraints to demonstrate their relationship:

- Running shuffling tasks is not greater than maximum shuffling tasks at any time.
- Running sort-computing tasks is not greater than maximum sort-computing tasks at any time.
- The total running reduce tasks is not greater than the maximum reduce tasks at any time.

The admission constrains are performed under the collaboration of TaskTracker and ReduceTask. The TaskTracker, is responsible for asking for a reduce task from JobTracker to enter into a shuffle phase if both the running shuffling reduce task number and the total running reduce tasks number are smaller than the corresponding maximum ones. And it should also keep a record on the running sort-computing tasks number so that at the time when a reduce task finishing its shuffle phase enters a sort-compute phase it can decide whether this reduce task is allowed to go on if both the running sort-computing tasks number and total running number are less than the corresponding maximum ones. As for a reduce task, it will ask for the permission from TaskTracker when moving forward to next sort-computing phase. If not it would place this reduce task into the waiting list otherwise it looks into the waiting list first to put the head in the waiting list into execution and let this one wait until a new free compute slot is released. If the waiting list is null, this reduce task could get permission to step into the next round.

Here we have to set three different parameters rather than one in raw version Hadoop. So how do we decide the values? We put forward a formula which is apply to both maximum shuffling tasks slot and maximum sort-computing tasks slot based on the statistic information. As for the maximum total tasks slot, it is set to be the larger one of the above two. P stands for the total number of reduce task, x represents the maximum reduce tasks number allowed running simultaneously in raw hadoop mapred.tasktracker.reduce.tasks.maximum and Avg_x is the average running time in raw version. It is known that the running time of a reduce task will be shorter if the parameter mapred.-tasktracker.reduce.tasks.maximum is smaller. So we set $Gain_x$ to be the total gain of a job in reduce phase in raw version if mapred.tasktracker.reduce.tasks.-maximum set to be x and $Lose_x$ to be the total lose of a job in a reduce phase. Here we have two values for mapred.-tasktracker.reduce.tasks.maximum m and n where m is smaller than n (m<n). So $Gain_m$ and $Lose_m$ is represented as follows:

$$Gain_m = (Avg_n - Avg_m) * P/n \tag{13}$$

$$Lose_m = (P/m - P/n) * Avg_m \tag{14}$$

If $Gain_m > Lose_m => Avg_n/n > Avg_m/m$, that means we set mapred.tasktracker.reduce.tasks.maximum to be m is better than n and vice versa.

Although our admission constraint in this paper is implemented explicitly in terms of slot, its inner mechanism is based on evaluation to different resources utilization. This evaluation is static and determined by user configuration. A dynamic real-time resource utilization detection can be used to determine when to launch a shuffle reduce task or a compute reduce task which we will cover in the future work section.

Table 1: Hardware configuration list

#nodes	11
#CPU in each node	4
#core in each CPU	1
CPU	AMD 1.8GHz
Memory in each node	5.9G
Disk in each node	186G
Network	Gigabit

6. EVALUATION

We set up two benchmarks to evaluate our techniques in resources utilization efficiency, one of which is a microbenchmark in that benchmark every technique is tested one by one and the other macrobenchmark showing the effects of all techniques put together. At the same time we implement two workloads. A single job workload used to test resource pipeline comes from gridmix2 which is a standard benchmark in Hadoop and a multi-job workload showing the results of multi-job scheduling as well as the overall techniques is also a variant of gridmix2. Our single job workload is called monsterquery which is a configurable job where map tasks output some keepMap percentage of input records and reduce tasks output some keepReduce percentage of the intermediate records. We also set a new parameter called keepCompute which represents how many mathematic operations we want to perform before processing one map input record. So we can extend monsterquery into a CPU-intensive, IO-intensive job or mix job through these parameters. A mutil-job workload, which contains a CPU-intensive monsterquery, an IO-intensive monsterquery and a mix monsterquery, is used to test mutil-job status scheduling. And we submit these jobs by a time interval.

Our benchmarks are performed in the environment shows in table 1:

The cluster is configured in one rack. Operating system running on it is CentOS release 5.3, Linux version 2.6.18-128.el5, Apache Hadoop version 0.20.2 and JDK version 1.6.0_14.

Definition: throughput (T) is the number of jobs finished in unit time. Suppose: we finished n jobs in time t, so we get:

$$T = \frac{n}{t} \tag{15}$$

In the test, we will use the same workload both in the raw version Hadoop and the new version Hadoop, So:

$$n_{raw} = n_{new} \tag{16}$$

Suppose we used time t_{raw} to represent the running time in the raw version Hadoop, and time t_{new} in the new version Hadoop. Then we get the throughput increased by I:

$$I = \frac{T_{new} - T_{raw}}{T_{raw}} = \frac{n_{new}/t_{new} - n_{raw}/t_{raw}}{n_{raw}/t_{raw}} = \frac{t_{raw}}{t_{new}} - 1 \tag{17}$$

Definition: wall time is the total time from the job submitted to it finished. Suppose the wall time of job i is t_1 in the raw version, and t_2 in the new version. And the percentage of wall time is reduced by r_i, which we also call single job wall time gain. Then:

$$r_i = \frac{t_1 - t_2}{t_1} \tag{18}$$

Suppose the average wall time gain for all jobs in the workload is r_{ave}. Then:

$$r_{ave} = \frac{1}{n}\sum_{i=1}^{n} r_i \quad (19)$$

In the test, we will record and evaluate the resource utilization using user CPU, system CPU iowait CPU,idle CPU,svctm,await and network IO whose meaning are the same with LINUX command iostat.

6.1 Microbenchmark

6.1.1 Impact of Map slot and job type on performance

We generate a data input set of 27G using TextWriter and run a monsterquery job including 200 map tasks and 100 reduce tasks with 128mb block size against it to test the impact of the parameter mapred.tasktracker.map.tasks.maximum on performance. We know in section 5.1 that this parameter in the new version Hadoop has a different meaning from the raw version Hadoop. So we want to know how the running time change as this parameter varies. And we also tune the parameters keepMap and keepCompute in monsterquery to make this job's map phase into three types: CPU-intensive, IO-intensive and mix one so that we can evaluate the effect of our technique on different type jobs.

Figure 6 shows the normalized running time of different map slot and job type in new version over the raw version. The ratio of this parameter ranges from 2-2 to 4-4. Firstly task pipeline in map phase produces a more obvious performance gain in mix type job and IO-intensive job than the CPU-intensive job. Figure 5 in section 5.2.2 tells us that if the first round additional map task is CPU-intensive that means it will stay in CPU phase for a long time and a second round normal CPU-intensive map task will have CPU resource conflict with it. In the future we will judge the map task type in advance using some statistics such as CPU period duration time, output data volume and output time. Then schedule different type map tasks to the same node avoiding single resource competition.

Secondly the ideal point in new version Hadoop is the time when map total slot is configured to 6 in IO-intensive job or 8 in mix type job, that is to say, the ratio of normal slot and additional slot is 3 to 3 or 4 to 4. At present we do not have some mathematic methods to judge the best ratio of map slot because map task running time is shorter compared to reduce task. In the future we will apply a dynamic configuration policy to every slave node which allows map slot ratio in a single node to change separately according to that node's ratio of map task. For example if the IO-intensive tasks in a node take a larger proportion the ratio will be set to 3:3. If MIX type task is the major part or IO-intensive and CPU-intensive is almost equal to each other after running for a period it will change from 3:3 to 4:4.

6.1.2 Compare in resource utilization of map phase

Now we choose IO-intensive monsterquery job with map slot configuration 6 in new version Hadoop and 4 in raw version Hadoop to investigate our resource utilization improvement. We *userCPU* utilization to measure resource usage efficiency since IO utilization can also be expressed by *sysCPU* and *iowaitCPU*.

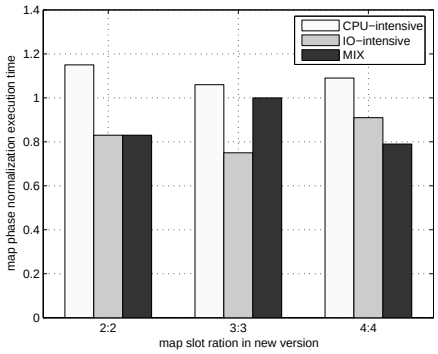

Figure 6: impact of job type and map slot ratio on performance compared to raw version whose map slot is set to 4

Table 2: CPU utility percent in map phase

	user CPU	sys CPU	iowait CPU	idle CPU
raw version	87.69	6.43	0.81	5.06
dynamic buffer	91.09	5.08	0.69	3.15
new version	93.71	4.00	0.22	2.07

From table 2 we learn that the system cpu percentage in raw version is higher than dynamic-buffer version and new version because it has more IO cost due to its intermediate spills and sorts. Whereas the dynamic-buffer version has lower user cpu percentage compared to new version since when a round of map tasks all stay in IO period no any additional map task runs in CPU period which gives rise to the cpu resource underuse problem.

Another resource we need to focus on is memory. Since memory consumption in new version Hadoop is larger than the raw one. Figure 7 shows under the same output size 128M of every map different JVM configuration has different execution time using the new version, the larger the faster. But the worst situation is the same with the raw one because it can switch to the raw version process logic at run time if the free memory is not enough.

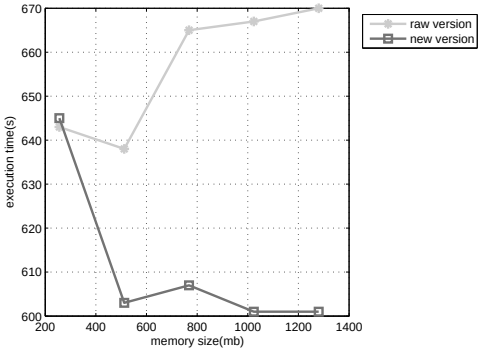

Figure 7: Effect with memory size change ranging from 256 to 1280

6.1.3 Reduce phase performance test

According to the formula listed in section 4.2 experiment 1 is designed to compute the best shuffle slot number and

Table 3: The execution result with different params

	avg-shuffle time (s)	avg-compute time (s)	total-running time(s)	shuffle Gain	shuffle Lose	compute Gain	compute Lose
2	12	28	1078	18	36	12	84
3	15	30	982	15	15	15	30
4	18	33	959	28	18	12	33
5	25	36	959	27	25	36	36
6	33	46	962	-	-	-	-

Table 4: CPU utility percent in reduce phase

	user CPU	sys CPU	iowait CPU	idle CPU
raw version	72.05	8.95	4.00	14.95
new version	79.22	11.44	2.36	6.96

compute slot number. We run 5 different configuration monsterquery jobs and the table 3 is the execution result.

We pick up 3 as the optimal number of concurrently shuffle and 5 concurrently compute. The raw version configuration is map slot 4, reduce slot 4 which is the best situation in our cluster, and the new version configuration is map slot 4, reduce total slot 5, reduce shuffle slot 3, reduce compute slot 5 or reduce shuffle slot is 4. We use example job in experiment 1 to test the effect. Figure 8 shows normalized running time of the new version Hadoop over the raw version. Our new version Hadoop has a better execution performance though its average copy time and average compute time is longer than raw one since we could schedule more reduce tasks in a round and resource usage complementary is more obvious. Table 4 reveals the user CPU utilization percentage between raw and new from which we can see the average user CPU usage of new version is higher than the raw one. But the raw version outperforms the new version in the term of the system CPU usage percentage. That is because in the new one the operating system needs to schedule more processed than the raw one which has a higher cost in system call.

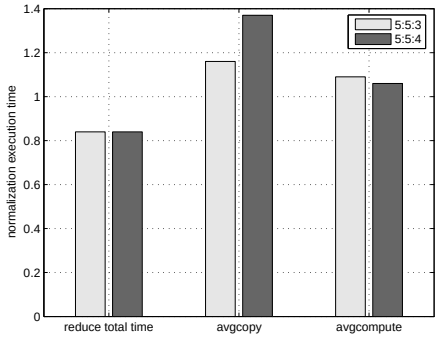

Figure 8: Running time of different reduce slot ration in new version compared to raw version which is set to be 1

6.1.4 Dynamic Resource Allocation test

We pick up a mutil-job workload, because we will nerver met the "Reduce Slot Hoarding" problem in a single job workload, to test the performance of dynamic resource allocation and use FairScheduler as a comparison. The workload contains a CPU-intensive monsterquery (job1), which has lots of calculation in each task (map task and reduce task), and requires a lot of CPU resource; An IO-intensive monsterquery (job2), which has a few calculation but needs to read and transfer lots of data, so needs IO resource; And a mix monsterquery (job3), which needs both CPU resource and IO resource. We generate 18G data for each job in the workload, and submit each job by a time interval which means if we have submitted the job1, we will submit the job2 after a time interval, and the job3 the next time interval. We submit jobs like this to simulate the real environment, because different users will submit jobs at different times, even for one user, he will have different needs at different times. We will submit a big job first to reproduce the "Reduce Slot Hoarding" problem. The figure 9 shows the execution result:

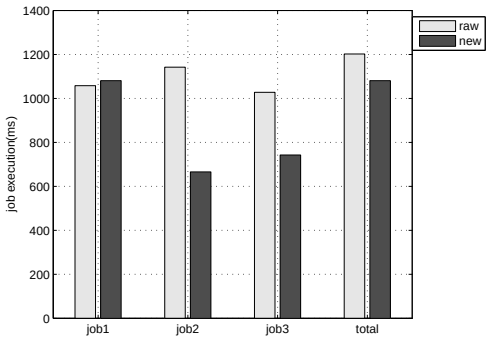

Figure 9: The execution time of each job

Figure 9 shows the execution time of each job and the total time we finish the whole workload. We set the time interval to 60 s. In the new version Hadoop, the job1 is a little slower, but the others jobs are much faster than the raw version Hadoop. And the total time is less. In the test, the total time we spend in the raw version Hadoop is 1202s, in the new version Hadoop is 1081s, so:

$$I = \frac{t_{raw}}{t_{new}} - 1 = \frac{1202}{1081} - 1 \approx 11.2\%$$

So, the average throughput is increased by 11.2%. And we can get:

$$r_{ave} = \frac{1}{3}\sum_{i=1}^{3} r_i = 22.4\%$$

So the average wall time gain is 22.4%. We also recorded the CPU, Disk I/O and Net I/O resource, as showed in table 5, figure 10, and table 6.

From the result, we can get that the CPU resource allocated is more reasonable in the new version Hadoop. Because the user CPU is 5.9% higher than the raw version

Table 5: The CPU utility percent in dynamic resource allocation test

	user CPU	sys CPU	iowait CPU	idle CPU
raw version	78.4	5.4	5.2	11
new version	84.3	4.9	1.5	9.3

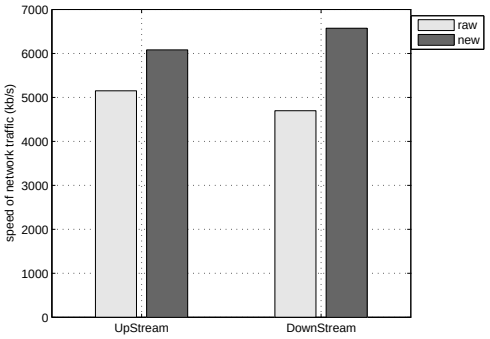

Figure 10: The Net I/O in dynamic resource allocation test

Table 6: The bottleneck of Disk I/O in dynamic resource allocation test

	await	svctm
raw version	65	21
new version	28	21

Table 7: The macrobenchmark

Job size	Big	Middle	Small
input	25G	10G	2.5G
type	CPU/IO	CPU/IO/Mix	CPU/IO/Mix
number	2	3	5

Hadoop, and the iowait CPU is 3.7% less than the raw version Hadoop. For the Net I/O resource, the upstream speed and downstream speed are increased by 18.1% and 39.9%. What's more, in the new version Hadoop, the parameters await and svctm are 56.9% closer than the raw version Hadoop, so we have relieved the Disk I/O bottleneck.

6.2 Macrobenchmark

We ran a mutil-job benchmark based on monsterquery job. The benchmark used 10 monsterquery jobs with different size and *keepMap*/*keepReduce*/*keepCompute* values,as show in table 7. We submitted these jobs by a time interval liked testing the Mutil-job Status Scheduler.

Figure 11 shows the execution result:

From figure 11, we get:

$$I = \frac{t_{raw}}{t_{new}} - 1 = \frac{1457}{1197} - 1 = 21.72\%$$

$$r_{ave} = \frac{1}{10}\sum_{i=1}^{10} r_i = 55.83\%$$

So we have increased the throughput rate by 21.74% and reduced the waiting time by 55.83%. We also recorded the use of resources as shown in table 8, table 9, and figure 12. From these, we can get that the CPU resource allocated is more reasonable in the new version Hadoop. Because the user CPU is 12.93% higher, the *iowaitCPU* is 6.61% less, and the idle CPU is 6.73% less than the raw version Hadoop. For the Net I/O resource, the upstream speed and downstream speed are increased by 11.3% and 23.5%. What's more, in the new version Hadoop, the parameters *await* and *svctm* are 30.3% closer than the raw version Hadoop, so we have relieved the Disk I/O bottleneck.

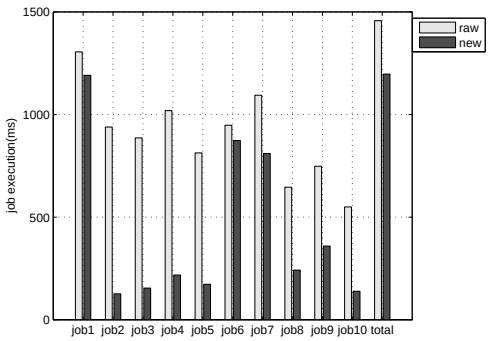

Figure 11: The time we executed the benchmark

Table 8: CPU utility in the dynamic resource split test

	user CPU	sys CPU	iowait CPU	idle CPU
raw version	70.82	10.16	10.87	8.15
new version	83.75	10.57	4.26	1.42

Table 9: The bottleneck of Disk I/O in the dynamic resource split test

	await	svctm
raw version	127	32
new version	96	35

7. RELATED WORK

MapReduce simplifies the programming of many parallel applications. Currently there are several MapReduce implementations available based on the Google's MapReduce architecture and some of these have improvements over the initial model. Through user side configuration as well as system automatic load balance based on the resource detection information, some MapReduce implementation such as hadoop can implement coarse granularity resource control which guarantees that every slave node in the cluster can have nearly the same resource assumption. The resource usage unbalance of several slave nodes in a large scale cluster has a smaller effect than in a small scale cluster where we need a finer granularity resource control to make sure resource on every slave node is utilized efficiently. However for our knowledge there are no other implementations that support features such as single node resource overlap or slot assignment policy to support resource usage efficiently as in our new version hadoop.

Phaser is a new coordination construct that integrates point-to-point and collective synchronization in the presence of dynamic parallelism by giving each activity the option of registering with a phaser in a signal-wait mode for barrier synchronization. So all processes running in phaser cannot go on at a certain point until other partner processes get to

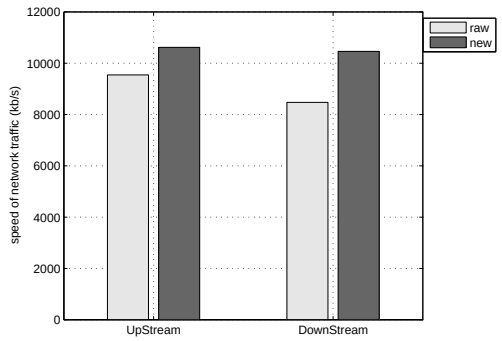

Figure 12: The Net I/O in the dynamic resource split test

the same point.And phaser accumulator is an advancement to phaser for reduction. It separates reduction into the parts of sending data, performing the computations and retrieving the results enabling overlap of communication and computation to exploit the CPU resource as much as possible.

Hadoop is an open source MapReduce implementation which is already introduced in section 2 Sphere is a parallel runtime that operates on Sector distributed file system. It can execute MapReduce style computation. Another parallel runtime Dyradthat supports Directed Acyclic Graph (DAG) based on execution flows provides more parallel topologies compared to MapReduce programming model. Above three implementations have a common point that is resource usage is sequential either within a task or between two tasks which inevitably causes resource utilization conflict or idle problems.

8. CONCLUSION AND FUTURE WORK

In this paper we discussed our experience in designing and implementing a new version Hadoop which has a dynamic slot assignment policy globally as well as a finer granularity resource utilization control locally. We also introduced the resource usage efficiency in raw Hadoop in section 2. What we want to prove is the resource utilization efficiency in raw version Hadoop does leave something to be desired. The overall performance gain is around 20% using our techniques. In this paper we used different phases' CPU average utilization percentage to measure the resource usage efficiency of Hadoop system since whatever IO resource or slot resource can be reflected in terms of CPU. We have also presented the results of a set of applications with voluminous data sets. The results indicate that new version Hadoop does perform well whatever in single job test of multi-job test. We plan to extend our future research in three areas: (i) Research on dynamic assignment of reduce task based on the real-time resource detection information which includes two aspects: when a JobTracker assigns a new submitted reduce task into shuffle period and when a TaskTracker assigns a waiting reduce task into compute period; (ii) Explore the possible schedule policy based on map task type so that the IO-intensive map task can complement with CPU-intensive map task to reduce CPU usage conflict mentioned in section 6.1.1 and map slot ratio can be changed separately in every single node rather than a static configuration by user; (iii) Monitor various resource of cluster, and forecast the resource, which needed by the map tasks and reduce tasks of each job. Then we can fully automatic allocate resource without define how many slots in the cluster. With the above enhancements, the new version Hadoop will provide a real-time dynamic resource detection and schedule policy under the cooperation of JobTracker and TaskTracker to maximize the resource utilization of a cluster.

9. ACKNOWLEDGEMENTS

We would like to thank Jian Lin, Xicheng Dong and Maosen Sun for their guideline and feedback on this paper, and Chao Tian for his contributions to the early scheduler. We are grateful to all large-scale data processing team members for their contributions used in this paper. What's more, this paper is partially supported by the National Natural Science Foundation of China (grant No. 60873243), the National High-tech R&D Program of China (grant No. 2010AA012502, 2010AA012503), and the Hi-Tech Research and Development (863) Program of China (Grant No. 2011AA01A203).

10. REFERENCES

[1] Hadoop. http://hadoop.apache.org/.

[2] R. E. Bryant. Data-intensive supercomputing:the case for disc. Technical report, School of Computer Science Carnegie Mellon University Pittsburgh, PA 15213, May 10, 2007.

[3] T. Condie, N. Conway, P. Alvaro, J. M. Hellerstein, K. Elmeleegy, and R. Sears. Mapreduce online. Technical report, UC Berkeley Yahoo! Research, 2009.

[4] J. Dean and S. Ghemawat. Mapreduce: Simplied data processing on large clusters. In *OSDI'04: Sixth Symposium on Operating System Design and Implementation*, 2004.

[5] Y. Gu and R. Grossman. Sector and sphere: The design and implementation of a high performance data cloud. *Theme Issue of the Philosophical Transactions of the Royal Society A: Crossing Boundaries: Computational Science, E-Science and Global E-Infrastructure*, 367, 2009.

[6] K. Huang and Z. Xu. *Scalable Parallel Computing: Technology, Architecture, Programming*. McGraw-Hill, Inc., 1998.

[7] M. Isard, M. Budiu, Y. Yu, A. Birrell, and D. Fetterly. Dryad: Distributed data-parallel programs from sequential building blocks. In *European Conference on Computer Systems (EuroSys)*, 2007.

[8] J. Shirako, D. Peixotto, V. Sarkar, and W. Scherer. Phaser accumulators: A new reduction construct for dynamic parallelism. In *IPDPS*, 2009.

[9] J. Shirako, D. M. Peixotto, V. Sarkar, and W. N. Scherer. Phasers: a unified deadlock-free construct for collective and point-to-point synchronization. In *ICS'08*, 2008.

[10] M. Zaharia, D. Borthakur, J. S. Sarma, K. Elmeleegy, S. Shenker, and I. Stoica. Job scheduling for multi-user mapreduce clusters. Technical report, University of California, Berkeley Facebook Inc Yahoo! Research, 2009.