A Throughput Driven Task Scheduler for Batch Jobs in Shared MapReduce Environments

Xite Wang, Derong Shen, Ge Yu, Tiezheng Nie, Yue Kou
College of Information Science and Engineering, Northeastern University, P.R. China
wangxite@research.neu.edu.cn, {shenderong, yuge, nietiezheng, kouyue}@ise.neu.edu.cn

Abstract
MapReduce is one of the most popular parallel data processing systems, and it has been widely used in many fields. As one of the most important techniques in MapReduce, task scheduling strategy is directly related to the system performance. However, in multi-user shared MapReduce environments, the existing task scheduling algorithms cannot provide high system throughput when processing batch jobs. Therefore, in this paper, a novel scheduling technique, Throughput-Driven task scheduling algorithm (TD scheduler) is proposed. Firstly, based on the characteristics of shared MapReduce environments, we propose the framework of TD scheduler. Secondly, we classify the jobs into six states. Jobs in different states have different scheduling priorities. We also give the rules of state conversion, which can ensure the fairness of resource allocation and avoid wasting system resources. Thirdly, we design the detailed strategies for job selection and task assignment. The strategies can effectively improve the ratio of local task assignment and avoid hotspots. Fourthly, we show that our TD scheduler can be applied to the heterogeneous MapReduce cluster with small modifications. Finally, the performance of TD scheduler is verified through plenty of simulation experiments. The experimental results show that our proposed TD scheduler can effectively improve the system throughput for batch jobs in shared MapReduce environments.

Keywords: MapReduce; shared environment; batch job; task scheduling; throughput

1. INTRODUCTION

Nowadays, various of applications, such as internet access, business computing and scientific research, generate large amount of data continually. Thus, the analysis of big data becomes a very hot topic which attracts the attention of many researchers [1,2]. As a popular parallel data processing system, MapReduce [3] has been proven to be an efficient technique for big-data analysis, and it has been adopted in many fields. With the help of the MapReduce system, users can expediently process PB level of data without considering the execution details (e.g. data distribution, fault-tolerance).

Each MapReduce job contains several map and reduce tasks. Based on a specific scheduling strategy, the system will assign the tasks to the corresponding computing nodes and process them in parallel. Obviously, task scheduling is quite crucial for the system performance. An unbefitting task scheduler may cause lots of network transmission and reduce the system processing capacity.

In many practical applications, a large scale MapReduce cluster is often shared by several users. In a period of time, the cluster may receive a batch of jobs submitted by different users. Each job applies for a percentage of computing resources based its workload and the degree of importance, and all the jobs must be finished within the required time. Hence, in this shared MapReduce environment, the system needs a suitable scheduling strategy that can provide high throughput along with the fairness of resource allocation. In fact, the above problem is quite common. For example, an online shopping company (e.g. Amazon) builds a large MapReduce cluster shared by many departments. In the period that the internet access is rare (e.g. midnight), these departments will submit many MapReduce jobs, such as access log analysis, transaction information collation etc. Each job will apply a proportion of computing resources, and the system must finish all these jobs within this period so that the daily work is not affected. In this case, the system must first improve the throughput to finish the jobs on time. Meanwhile, the fairness of resource allocation should also be considered to avoid that individual job monopolizes the system resources.

In order to solve the problem above, in this paper, we propose TD scheduler which is designed for improving the system throughput for batch jobs in a shared MapReduce environment. In many MapReduce applications, most of the calculations are processed in the map phase, and the map phase determines the time cost of the entire job. Thus, our TD scheduler is mainly designed for the map phase. The contributions of this paper are summarized as follows.

1. We state the task scheduling problem for batch jobs in a shared MapReduce environment, and propose the frame of TD scheduler.

2. We classify the jobs into 6 states and give the rules of state conversion, which can ensure the fairness of resource allocation and avoid the waste of computing resources.

3. We design the detailed strategies for job selection and task assignment, which can significantly reduce the network overhead and improve the system throughput.
4. Through a small amount of modifications, we show that our TD scheduler can be applied in the heterogeneous MapReduce cluster.

5. We verify the effectiveness of TD scheduler through plenty of experiments.

The rest of this paper is organized as follows. We overview the MapReduce framework and state several previous approaches of task scheduling as background knowledge in Section 2. We define the problem model and state the design objective in Section 3. We describe the TD scheduler in detail in Section 4. We extend the TD scheduler to the heterogeneous environment in Section 5. We evaluate the performance of TD scheduler in Section 6. Finally, the conclusion of this paper is stated in Section 7.

2. BACKGROUND KNOWLEDGE

In this section, we firstly overview the MapReduce framework. Then we review the related work.

2.1 MapReduce Overview

The MapReduce system is presented by Google in 2004. It is designed for the large-scale data processing in a cluster. In general, the input files are stored in a distributed file system, and a task is submitted to the map or reduce phase, each mapper reads a specified pair of key-value and outputs a new pair of key-value as intermediate result. Then according to the partition function, all the intermediate pairs with the same key are grouped into a sequence and sent to the corresponding reducer. In the reduce phase, each reducer firstly merges the received pairs and generates new pairs of key-value, then calls the reduce function and outputs the final results.

The task assignment of MapReduce follows the data locality principle. When a slot requires task, the local task will be allocated preferentially. If no local task exists, the scheduler choose a task which is close to the slot in the network in order to induce the network overhead.

2.2 Related Work

As the kernel technique of parallel data processing, task scheduling has been studied for years. There have been many excellent task schedulers in MapReduce[5,8].

The default task scheduler of the Hadoopt system[9], Job Queue Task scheduler (also called FIFO scheduler) can provide a fundamental and effective method for task assignment. It sorts the jobs in the order of the submission time. Each time only one job can be scheduled in the system, and the later jobs will not be scheduled until all the former jobs are finished.

Some schedulers are designed for multi users to safely share a large MapReduce cluster. The Capacity scheduler[10] allocates computing resources to different users according to the configuration, which can avoid individual user monopolizes the system. It supports multiple job queues, and each queue is associated with a percentage of system resources. In each queue, the jobs are scheduled by FIFO. If some resource is idle, they will be shared with the other queues. Fair scheduler[11] is also used for the cluster shared by multi users, and the design objective is "fairness". Specifically, it supports several job pools and each pool is constituted by some jobs with specific weights. Fair scheduler assign computing resources to each job according to its weight. The idle resources will be allocated to the job with the most vacancies. Based on Fair scheduler, Matei Zaharia et al. presented delay scheduling[12,13] which can further improve the data locality. Furthermore, Thomas Sandholm et al.[14] presented a scheduling strategy which supports dynamic resource allocation. The scheduler allows users to apply for more computing resources if the jobs they submit are important.

In practice, the performance of the nodes in a cluster is different, the computing power of some old machines may be much lower than the new ones. Therefore, Matei Zaharia et al. presented the LATE Scheduler[15] for the heterogeneous environment. The scheduler estimates the processing ability of each node based on the accomplished tasks. If a task is executed slower than the others, it will be restarted on the node with good processing capacity. YongChul Kwon et al. discussed the skew problem in MapReduce and presented skewtune[16]. The technique detects the skew by estimating the finish time of tasks. The rest of a slower tasks will be split and assigned to the other nodes to be executed in parallel.

Besides, several studies focus on task scheduling for MapReduce jobs with deadlines. Jorda Polo et al. presented PD scheduler[17] applying to the deadline scheduling. Each job is associated with a specific deadline. According to the accomplished tasks, the scheduler calculates the expected finish time of the job. If a job cannot finish before the deadline with the allocated resources, the idle resources will be allocated to it preferentially. Kamal Kc et al.[18] presented a job execution cost model to estimate the finish
time of a job. A job will be accepted if the expectant finish time meets the deadline. Otherwise, the job will be rejected.

In brief, the previous studies rarely consider the system throughput problem for batch jobs in shared MapReduce environments. In contrast, our TD scheduler can provide better throughput and has strong practicability.

3. PROBLEM STATEMENT

In this paper, we mainly focus on the system throughput problem in MapReduce. Firstly, the environment can be formalized as follows. For a MapReduce cluster with \( T \) task slots, the jobs submitted by different users form a job queue \( \mathcal{Q} = \{ j_1, j_2, \ldots, j_n \} \). For each job \( j \) in \( \mathcal{Q} \), the related input file is stored in DFS, and \( j \) is associated with a parameter \( j.demand \) which means the number of task slots that \( j \) expects to occupy. \( j.demand \) can be transformed to other common parameters (such as job weight, deadline, etc.) easily, and it has good generality.

In order to provide good throughput, the system adopt the threshold method to ensure the fairness of resource allocation. Given a user specified lower bound \( L (0 < L < 1) \), the system guarantees the number of task slots that a running job \( j \) occupies: \( j.occupied \geq L \times j.demand \). Given a user specified upper bound \( H (H > 1) \), the system guarantees the number of task slots that a running job \( j \) occupies: \( j.occupied \leq H \times j.demand \), which can avoid individual job monopolizes the computing resources.

Thus, given a MapReduce cluster with \( T \) slots and a job queue \( \mathcal{Q} \) with \( n \) jobs, under the premise that all the running job satisfy the constraints of \( L \) and \( H \), the goal of our TD scheduler is to effectively improve the system throughput so that all the jobs in \( \mathcal{Q} \) can finish as quickly as possible.

4. TD SCHEDULER DESCRIPTION

In this section, we describe the details of TD scheduler. Firstly, we give the framework of TD scheduler. Then, we introduce the job states. At last, we describe the method of task assignment.

4.1 Frame of TD Scheduler

![Figure 2. Frame of TD Scheduler](image)

In general, there are 3 basic states for the jobs in \( \mathcal{Q} \). a) waiting, none of the task of the waiting job has been scheduled, and the set of the waiting jobs is denoted as \( S_{wait} \) (for simplicity, we assume that all the jobs in \( S_{wait} \) have been sorted in the order of submission time). b) running, only the running jobs need to be scheduled, and the set of the running jobs is denoted as \( S_{run} \). c) completed, all the tasks of the completed job have been finished, and the set of the completed jobs is denoted as \( S_{com} \).

Figure 2 describes the frame of TD scheduler. When a slot becomes idle and it requests a task, a task assigned to the slot. Firstly, the scheduler considers the jobs in \( S_{run} \) and choose a job (e.g. \( j_2 \)). Then, an untreated task of this job will be selected by TD scheduler and assigned to the slot.

4.2 Job States

According to the frame of TD scheduler, only the jobs in \( S_{run} \) need to be scheduled. However, in order to ensure the allocation fairness for the running jobs and make full use of computing resources, basic job classification is not enough. Hence, we introduce 4 new job states in this section.

In the initialization phase, all the jobs in \( \mathcal{Q} \) are in state waiting, \( S_{wait} = \mathcal{Q} \). All the \( T \) slots are idle and running job set \( S_{run} \) is empty. Intuitively, the first \( k \) jobs of \( S_{wait} \) can be inserted into \( S_{run} \) if

\[
\sum_{i=1}^{k} j.demand \leq T
\]

Along with the processing, some of the jobs in \( S_{run} \) are finished and inserted into the completed job set \( S_{com} \). Then some jobs in state waiting join into \( S_{run} \). The updating of \( S_{run} \) follows the principle below.

**PRINCIPLE 1** For \( \forall j \in \mathcal{Q} \) (\( j > 1 \)), \( j \) can join into \( S_{run} \) only if \( j \in S_{run} \).

In order to ensure the allocation fairness and high utilization rate of computing resources, based on the main parameters of a job (including the number of slots \( j \) occupies \( j.occupied \), the number of slots \( j \) expects \( j.demand \), the number of untreated tasks of \( j \) \( j.remain \)), we classify the running jobs into 4 types.

As Figure 3 shows, there are 4 states for the running jobs, including infantile, teenaged, adult and senile. The corresponding job sets are denoted as \( S_{inf}, S_{teen}, S_{adult}, S_{sen} \). The jobs in different states have different scheduling priorities. The states convert only follows the arrows. Next, we describe the states in more detail.

![Figure 3. Job States](image)

**Infantile** is a specific job state, and for all the running jobs, at most 1 job is in state infantile. The purpose of state infantile is to keep that all the slots can obtain tasks and...
avoid resource wasting. The job in state infantile is not restrained by the bounds and has low scheduling priority.

Theoretically, when the running jobs cannot occupy all the slots, some slots may get no task to run. In order to avoid resource wasting, the first job in \( S_{\text{init}} \) will join into \( S_{\text{lf}} \) for high system utilization. Formally, a waiting job \( j_{\text{wait}} \) can join into \( S_{\text{lf}} \) if

\[
\begin{align*}
D + \sum_{j \in S_{\text{lf}}} j_{\text{occupied}} < T \\
j_{\text{wait}}_{\text{demand}} > T - D - \sum_{j \in S_{\text{lf}}} j_{\text{occupied}}
\end{align*}
\]

where \( D = \sum_{j \in S_{\text{run}} \cup S_{\text{adu}}} j_{\text{demand}} \).

**Teenaged** is the beginning state for a running job \( j \), which means that the system already has enough slots for \( j \) (more than \( j \) expects), but actually the number of slots that \( j \) has occupied does not meet the constraint of lower bound (less than \( L \times j_{\text{demand}} \)). In order to ensure the allocation fairness, the jobs in state teenaged have the highest priority.

With the jobs in \( S_{\text{run}} \) finishing gradually, the system obtains more slots to reallocate. For an infantile job \( j_i \), if the number of slots that the system can reallocate is more than \( j_i_{\text{demand}} \), but the number of slots that \( j_i \) occupies does not meet the constraint of lower bound, \( j_i \) will be converted to a teenaged job. Formally, \( j_i \) can join into \( S_{\text{teen}} \) if

\[
\begin{align*}
j_{i, \text{demand}} &\leq T - D - \sum_{j \in S_{\text{lf}}} j_{\text{occupied}} \\
j_i_{\text{occupied}} &< L \times j_i_{\text{demand}}
\end{align*}
\]

**Adult** is the general state for a running job. The number of slot that an adult job is stable and meets the constraints of both the lower bound and the upper bound. The jobs in state adult have the normal scheduling priority.

For an infantile job \( j_i \), if the number of slots that the system can reallocate is more than \( j_i_{\text{demand}} \), and the number of slots that \( j_i \) occupies satisfies the constraints of lower bound and upper bound, \( j_i \) will join into \( S_{\text{adu}} \). Formally, \( j_i \) can join into \( S_{\text{adu}} \) if

\[
\begin{align*}
j_i_{\text{demand}} &\leq T - D - \sum_{j \in S_{\text{lf}}} j_{\text{occupied}} \\
L \times j_i_{\text{demand}} &\leq j_i_{\text{occupied}} \leq H \times j_i_{\text{demand}}
\end{align*}
\]

If the number of slots that a teenaged job \( j_i \) occupies reaches \( L \times j_i_{\text{demand}} \), \( j_i \) can be promoted to an adult job. The conversion condition is similar to Equation 4, we do not give the unnecessary details.

**Senile** is last state for a running job. The job in state senile still has tasks in processing, but has no untreated task. The running tasks of a senile job consume some system resources, but do not need to be further scheduled or meet the constraint of lower bound.

When all the tasks of an adult job \( j_a \) have been assigned to the slots, \( j_a \) will join into the senile job set \( S_{\text{sen}} \). The conversion condition is

\[
\begin{align*}
j_a_{\text{remain}} &= 0 \\
j_a_{\text{occupied}} &\neq 0
\end{align*}
\]

For a senile job \( j_s \), along with the accomplishment of the running tasks, \( j_s \) finishes and it will join into the completed job set \( S_{\text{com}} \). When all the jobs in \( Q \) join into \( S_{\text{com}} \), the process completes.

Thus far, we have introduced all the job states and the conditions of state conversion. Next, Table 1 shows an example of job states.

As Table 1 describes, There is a MapReduce cluster with \( T=1000 \) slots, and a job queue \( Q=\{j_1, j_2, j_3, j_4, j_5\} \), lower bound \( L=0.7 \), upper bound \( H=1.3 \). In the initialization phase, according to Equation 1, \( j_1, j_2, j_3, j_4 \) can join into the running job set \( S_{\text{run}} \). At some point, there a slot becomes idle and requests task to run, and the processing details are described in case 1. \( j_1, \text{remain}=0 \) means that \( j_1 \) has no remaining untreated task and it is in state senile. \( j_2, j_3, j_4 \) are in state adult. Hence, according to Equation 2, the theoretical number of slots that running jobs can occupy is \((200+150+250)+399< 1000\), and \( j_3 \) is converted to state infantile. Then after several times of task assignments, the system receives a task request. If the situation is like case 2(a), according to Equation 3, the theoretical number of slots that the system can allocate to \( j_5 \) is \((200+150+250)-200\geq 200\), but the number of slots that \( j_5 \) occupies does not meet the lower bound, \( j_5, \text{occupied}=101 < 200 \times 0.7 \). Hence \( j_5 \) joins into the teenaged job set. If the situation is like case 2(b), according to Equation 4, \( j_5, \text{occupied}=143 > 200 \times 0.7 \). Hence \( j_5 \) is converted to state adult immediately.

### Table 1 Example of State Conversion

<table>
<thead>
<tr>
<th>Case</th>
<th>Parameters</th>
<th>( j_1 )</th>
<th>( j_2 )</th>
<th>( j_3 )</th>
<th>( j_4 )</th>
<th>( j_5 )</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>( j_{\text{demand}} )</td>
<td>400</td>
<td>200</td>
<td>150</td>
<td>250</td>
<td>200</td>
</tr>
<tr>
<td>2(a)</td>
<td>( j_{\text{occupied}} )</td>
<td>399</td>
<td>212</td>
<td>142</td>
<td>246</td>
<td>0</td>
</tr>
<tr>
<td></td>
<td>( j_{\text{remain}} )</td>
<td>&gt;0</td>
<td>&gt;0</td>
<td>&gt;0</td>
<td>&gt;0</td>
<td>&gt;0</td>
</tr>
<tr>
<td></td>
<td>( j_{\text{state}} )</td>
<td>senile</td>
<td>adult</td>
<td>adult</td>
<td>adult</td>
<td>wait - infantile</td>
</tr>
<tr>
<td>2(b)</td>
<td>( j_{\text{occupied}} )</td>
<td>200</td>
<td>248</td>
<td>165</td>
<td>285</td>
<td>101</td>
</tr>
<tr>
<td></td>
<td>( j_{\text{remain}} )</td>
<td>&gt;0</td>
<td>&gt;0</td>
<td>&gt;0</td>
<td>&gt;0</td>
<td>&gt;0</td>
</tr>
<tr>
<td></td>
<td>( j_{\text{state}} )</td>
<td>senile</td>
<td>adult</td>
<td>adult</td>
<td>adult</td>
<td>infantile - teenaged</td>
</tr>
</tbody>
</table>

### 3.2 Task Assignment

The task assignment contains 2 main steps. a) Select a suitable job from the running job set. b) Select a suitable task of the job. There are 2 types of assignments. a) Forced assignment is used to ensure the allocation fairness. b) Normal assignment is used for improving the system throughput. Next, we firstly introduce the 2 cases which trigger the forced assignment.

**CASE 1** When a slot finishes a task which belongs to job \( j \) and requires a new task to run, if \( j \in S_{\text{adu}} \cup S_{\text{teen}} \) and
j.occupied = j.demand - 1, we must assign a task of j to the slot in order to guarantee that j satisfies the constraint of the lower bound.

**CASE 2** If case 1 is not triggered but the teenaged job set \( S_{n<1} \neq \emptyset \), we must assign a task of the first job in \( S_{n<1} \) to the slot because the teenaged job has higher priority.

If the forced assignment is triggered, the specific job is selected. Then the task selection of this job follows the MapReduce default data locality principle (mentioned in Section 2.1). If the forced assignment is not triggered, the scheduler will assign the tasks normally, and the details are described as follows.

**A. How to select a job**

In order to maximize the system throughput, we mainly consider the following 2 principles for job selection.

**PRINCIPLE 2** High ratio of local processing can reduce the unnecessary network traffic.

In the parallel computing, data locality can significantly reduce the network overhead, and it is the most efficient way to improving the system throughput. However, the existing task schedulers for multi-user MapReduce clusters do not consider the data locality as the primary target. For example, the Fair scheduler assigns tasks mainly based on the job weight. Besides, the FIFO scheduler has considered the data locality, but it does not support multi-job co-scheduling and cannot be used in a shared cluster. In our TD scheduler, we preferentially select the job that has the local tasks, which can improve the throughput effectively.

We define the average processing time per task of job j as \( j.amt \), and it can be estimated easily based on the completed tasks of j. Then we give another principle.

**PRINCIPLE 3** For the local task assignment, the job which has smaller amt value has a higher priority to be selected. For the nonlocal task assignment, we choose the job with large amt value.

For the local task assignment, the task of the job with small amt value can be completed within a short time, and the slot can be released earlier, which can improve the flexibility of scheduling. For the nonlocal task assignment, because MapReduce adopts the streaming process mode, if we select a task of the job with large amt value, the delay ratio \(^1\) is small, which can indirectly improve the throughput. In addition, more tasks of the jobs with small amt values can be left for local assignment.

The principle can be explained through the following example. For job \( j_1 \), \( j_2 \), the split size of the input file is 128MB, and \( j_1.amt = 60s \), \( j_2.amt = 10s \). Hence, if we choose a task of \( j_1 \) for nonlocal assignment, the delay ratio is small if the network bandwidth can reach 128/60=2.13MB/s. For \( j_2 \), the bandwidth needs to reach 128/10=12.8MB/s. However, the network resources are limited in most of the distributed computing. We assume network bandwidth is 2MB/s. If we choose a task of \( j_1 \) for nonlocal task assignment, the shortest processing time is 128/2=64s, and the delay ratio of \( j_1 \) is (64-60)/64=0.063. If we choose a task of \( j_2 \), the shortest processing time is 64s, and the delay ratio is (64-10)/64=0.844. Hence, for a nonlocal task assignment, the task of \( j_1 \) (the job with larger amt value) can reduce the processing delay and improve the system throughput.

When a slot on node \( n \) requires task to the master, according to the principles above, the method of job selection can be summarized as follows.

**Step 1.** We attempt to make a local assignment. Firstly, we sort all the jobs in \( S_{n<1} \) according to the amt values and check them one by one. For a job j, if \( j \) has untreated tasks stored on node \( n \) and \( j \) satisfies the constraint of the upper bound (\( j.occupied < H \times j.demand \)), \( j \) is selected and a local task of \( j \) will be allocated. If there is no satisfied job in \( S_{n<1} \), we further check the infantile job. If the infantile job does not meet the condition, a nonlocal task assignment is unavoidable (step 2).

**Step 2.** If a nonlocal assignment is necessary, we sort the jobs in \( S_{n<1} \) according to the amt values in descending order. For each job \( j \), if \( j \) satisfies the constraint of the upper bound, \( j \) will be selected. If on adult job satisfies the constraint, the infantile job will be selected. Next, we introduce the method of selecting a suitable task of a given job.

**B. How the select a task**

After we select a specific job \( j \), we have already known whether the task assignment is local or not. If it is local, the method of task selection is straightforward and we do not give the unnecessary details. If we need to choose a nonlocal task, the network transmission is unavoidable. Basically, we intend to choose the task which is close to the slot in the network in order to reduce the network congestion. Furthermore, the hotspot problem is another enemy of the system throughput, and it cannot be ignored. Hence, we present a novel task selection method which can avoid hotspots.

In general, a hotspot is the node which transmits data with too many nodes simultaneously. In order to further analyze the problem, we classify the hotspots into two categories.

The node \( n \) is an actual hotspot if the total number of the nodes which transmit data with \( n \) is larger than the threshold \( \varepsilon \). In other words, given the network bandwidth \( \phi \), if a nonlocal task reads data from \( n \) across the network, the performance will be affected if the data transmission speed cannot reach to \( \phi.\varepsilon \).

To avoid actual hotspots, we employ a new data structure hot queue. a) Append operation. If there is a new nonlocal task assignment that a slot needs to read data from node \( n \) at time point \( t \), a tuple \( (n, t) \) will be appended to the end of hot queue. b) Remove operation. We denote the split size of the input file as \( \gamma \) and the network transmission speed as \( \phi.\varepsilon \). Periodically, we check the tuples of hot queue.

\(^1\) For a task \( t \), we denote the time cost of \( t \) as \( c_1 \) if \( t \) is processed locally, and \( c_2 \) if \( t \) is processed non-locally. Then the delay ratio of \( t \) is \( (c_2-c_1)/c_2 \).
in order. For each tuple \((n_t, t_i)\), if \(t_i + \varepsilon/\varphi \leq t_{cur}\), where \(t_{cur}\) is the current time, the tuple is overdue and we remove it from the hot queue. In addition, there is a corresponding statistical table. For each node \(n\) in the hot queue, the table records \(\text{count}(n)\), the total number of the tuples which contain \(n\), if \(\text{count}(n) = c\), we forbid more connections to \(n\) until some tuples which contain \(n\) become overdue.

Besides, node \(n\) is a potential hotspot if the remaining untreated workload on \(n\) is larger than that on the other nodes. Because in the foreseeable future, there is no local task on the other nodes, and lots of nodes need to read data from \(n\). The potential hotspot \(n\) becomes an actual hotspot.

To avoid potential hotspots, we firstly formalize the remaining untreated workload on node \(n\).

\[
R_n = \sum_{j=\text{teen} = S\text{sen}} j \cdot \text{amt} \times r_n^j
\]

where \(r_n^j\) is the number of the remaining untreated tasks of job \(j\) stored on node \(n\). Then the potential hotspot with the largest remaining workload has the highest probability to become an actual hotspot, and we intend to select the task on this node to reduce its remaining workload.

In summary, if the scheduler needs to select a nonlocal task of job \(j\) for the idle slot, we firstly get the nodes which satisfy the following 2 conditions. a) The node still has local tasks of \(j\). b) The current connection number of the node is smaller than \(e\) according to the statistical table. Then from these nodes, we choose the node which has the most remaining workload and assign a task of \(j\) stored on this node to the idle slot.

**Table 2 Example of Nonlocal Task Selection**

<table>
<thead>
<tr>
<th>node</th>
<th>connection number</th>
<th>number of tasks stored on node</th>
</tr>
</thead>
<tbody>
<tr>
<td>(n_1)</td>
<td>1</td>
<td>(r_{n_1}^{\text{inf}}=4, \ r_{n_1}^{\text{sen}}=5, \ r_{n_1}^{\text{adu}}=0)</td>
</tr>
<tr>
<td>(n_2)</td>
<td>3</td>
<td>(r_{n_2}^{\text{inf}}=5, \ r_{n_2}^{\text{sen}}=3, \ r_{n_2}^{\text{adu}}=4)</td>
</tr>
<tr>
<td>(n_3)</td>
<td>0</td>
<td>(r_{n_3}^{\text{inf}}=3, \ r_{n_3}^{\text{sen}}=0, \ r_{n_3}^{\text{adu}}=7)</td>
</tr>
<tr>
<td>(n_4)</td>
<td>0</td>
<td>(r_{n_4}^{\text{inf}}=3, \ r_{n_4}^{\text{sen}}=0, \ r_{n_4}^{\text{adu}}=0)</td>
</tr>
<tr>
<td>(n_5)</td>
<td>1</td>
<td>(r_{n_5}^{\text{inf}}=6, \ r_{n_5}^{\text{sen}}=4, \ r_{n_5}^{\text{adu}}=6)</td>
</tr>
</tbody>
</table>

Table 2 shows an example of nonlocal task selection. The cluster contains 5 slave nodes, and \(j_1, j_2, j_3\) are running in the system. \(j_1.\text{amt}=20s, \ j_2.\text{amt}=25s, \ j_3.\text{amt}=30s\). The connection threshold \(\varepsilon=3\). If an idle slot on node \(n_4\) requires a task, we select job \(j_1\) and have to make a nonlocal task assignment. Then, we firstly collect the nodes which still have untreated tasks of \(j_1\), \(\{n_2, n_3, n_4\}\). Because the connection number \(n_2\) of is equal to \(e\), we remove \(n_2\). Then we compute the remaining workload of \(n_3, n_5\), \(R_{n_3}=270, \ R_{n_5}=400\). Thus \(n_5\) is selected, and a task of \(j_1\) on \(n_5\) is assigned to the idle slot.

### 3.3 Algorithm Description

In this section, we illustrate the TD scheduler in more detail. Algorithm 1 describes the frame of TD scheduler. When a slot on node \(n\) finish a task of job \(j\), it sends a task request to the master, and then TD scheduler is invoked. First, update the number of slots that \(j\) occupies, \(j.\text{occupied}\) (line 1). If \(j\) is a teenaged or adult job and \(j.\text{demand xl}\), the forced assignment is triggered, and we assign a task of \(j\) to the slot (line 2.3). Otherwise, if \(j\) is a senile job, we need to update the job state (line 6-17). In particular, if \(j.\text{occupied}=0\), \(j\) is completed and we move it into \(S_{com}\) (line 7.8). If there is a infantile job \(j\), and \(j.\text{demand} \leq T-D-S_{\text{sen}}. j.\text{occupied}\), we need to move \(j\) into \(S_{\text{sen}}\) or \(S_{\text{adu}}\) based on Equation 3 and 4 (line 11-15). If there in no infantile job and \(S_{\text{sen}}. j.\text{occupied} + D < T\), we move the first waiting job into \(S_{\text{adu}}\) based on Equation 2 (line 16,17). Then if \(S_{\text{sen}}\) is not empty, the forced assignment is triggered, and we assign a task of the first teenaged job to the slot (line 18-20). Otherwise, we assign a task to the slot normally (line 21,22). The function \(\text{AssignTask Normal}(n)\) is introduced in Algorithm 2. At last, we update the parameters of the selected job and the job state (line 23-27).

**Algorithm 1 TD Scheduler**

**Input:** the node \(n\) which contains the idle slot, the job \(j\) which the task processed by the slot belongs to

**Output:** a task for the idle slot

01: \(j.\text{occupied}--;\)
02: if \((j \in S_{\text{adu}}) \cup S_{\text{sen}} \text{ and } j.\text{occupied} < j.\text{demand} \times L)\)
03: \(j_{\text{ass}} \leftarrow j;\)
04: \(\text{AssignTask SpecificJob} (j_{\text{ass}}, n);\)
05: else
06: if \((j \in S_{\text{sen}})\)
07: if \((j.\text{occupied}=0)\)
08: \(j\) joins into \(S_{\text{com}};\)
09: if \((S_{\text{inf}} \neq \emptyset)\)
10: get the infantile job \(j\), from \(S_{\text{inf}};\)
11: if \((j.\text{demand} \leq T-D-S_{\text{sen}}. j.\text{occupied})\)
12: if \((j.\text{occupied} < j.\text{demand} \times L)\)
13: \(j\) joins into \(S_{\text{sen}};\)
14: else
15: \(j\) joins into \(S_{\text{adu}};\)
16: else if \((S_{\text{sen}}. j.\text{occupied} + D < T)\)
17: move the first job in \(S_{\text{wait}}\) to \(S_{\text{inf}};\)
18: if \((S_{\text{sen}} \neq \emptyset)\)
19: \(j_{\text{ass}} \leftarrow \text{the first job in } S_{\text{sen}};\)
20: \(\text{AssignTask SpecificJob} (j_{\text{ass}}, n);\)
21: else
22: \(j_{\text{ass}} \leftarrow \text{AssignTask Normal} (n);\)
23: \(j_{\text{ass}}.\text{occupied} + +, j_{\text{ass}}.\text{remain} --;\)
24: if \((j_{\text{ass}} \in S_{\text{sen}} \text{ and } j.\text{occupied} < j.\text{demand} \times L)\)
25: \(j_{\text{ass}}\) joins into \(S_{\text{adu}};\)
26: if \((j_{\text{ass}}.\text{remain}=0)\)
27: \(j_{\text{ass}}\) joins into \(S_{\text{sen}};\)
28: end;

**Algorithm 2** describes the function \(\text{AssignTask Normal}(n)\), and the job selected by the function will be returned. Firstly, we sort all the jobs in \(S_{\text{adu}}\) according to the \(\text{amt}\) values (line 1). Then we check the jobs in \(S_{\text{adu}}\) in order. For each job \(j\), if it has untreated tasks stored on node \(n\) and satisfies the constant of the upper bound, we select \(j\) and the algorithm ends (line 2.5). If on job is selected, we check the
infantile job (if there is) with the same condition (line 6-9). If on job satisfies, we check the jobs in $S_{adv}$ in reverse order. For each job $j$, if it meets the constant of the upper bound, we select $i$ and select a node by using the function GetNonHotspotNode$(j)$ (will be described in Algorithm 3) (line 10-14). If no job meets the conditions, in order to avoid wasting resources, we assign a task of the infantile job to the slot (line 15-17).

Algorithm 2 AssignTask_Normal $(n)$

**Input:** node $n$ which the idle slot belongs to  
**Output:** assign a task of $j$ to the slot and return job $j$

1. sort the jobs in $S_{adv}$ according to the amt values;
2. for each $j$ in $S_{adv}$
3. if ($n$ has untreated tasks of $j$ and $j.occupied < j.demand*H$)
4. assign a local task of $j$ on $n$ to the slot;
5. return $j$;
6. get the infantile job $j$, from $S_{inf}$;
7. if ( $n$ has untreated tasks of $j$ and $j.occupied < j.demand*H$)
8. assign a local task of $j$, on $n$ to the slot;
9. return $j$;
10. for each $j$ in $S_{adv}$ in reverse order
11. if ($j.occupied < j.demand*H$)
12. $n_{ass} \leftarrow $ GetNonHotspotNode$(j)$;
13. assign a task of $j$ on $n_{ass}$ to the slot;
14. return $j$;
15. $n_{ass} \leftarrow $ GetNonHotspotNode$(j)$;
16. assign a task of $j$, on $n_{ass}$ to the slot;
17. return $j$;

Algorithm 3 GetNonHotspotNode $(j)$;

**Input:** the specific job $j$

**Output:** return a node which is not hotspot

1. job set $S$—get all the jobs which has untreated tasks of $j$;
2. remove the jobs whose current connection numbers are $e$ from these jobs (line 2).
3. at last, we select the job from the remaining jobs according to Equation 6 and return (line 3,4).

5. **EXTENDING TD SCHEDULER FOR HETEROGENEOUS ENVIRONMENTS**

In most of the researches, the authors assume that the MapReduce cluster is isomorphous for simplicity. However, in the real-world usage, it cannot be guaranteed that all the nodes in the cluster has the same computing capacity. In fact, there may be several generations of machines in a cluster, and the performance of these computers is different. In other words, the heterogeneous environment is very common.

Hence, in this section, we extend our TD scheduler for heterogeneous environments.

Firstly, we use a test method to quantify the computing capacity of a node in the heterogeneous cluster. In detail, we can run the same program with the same input on each node and record the processing time. The length of the processing time is a good measurement for the computing capacity. We denote the processing time on node $n_i$ as $t_i$, and average processing time as $\bar{t}$. Then the capacity factor of node $n_i$, $n_i cf = \frac{\bar{t}}{t_i}$, and we use capacity factor to measure the computing capacity of a node. Node with larger value of capacity factor has better computing capacity.

<p>| Table 3 Example of Capacity Factor |</p>
<table>
<thead>
<tr>
<th>node</th>
<th>processing time (s)</th>
<th>capacity factor</th>
</tr>
</thead>
<tbody>
<tr>
<td>$n_1$</td>
<td>10</td>
<td>1.3</td>
</tr>
<tr>
<td>$n_2$</td>
<td>12</td>
<td>1.08</td>
</tr>
<tr>
<td>$n_3$</td>
<td>15</td>
<td>0.87</td>
</tr>
<tr>
<td>$n_4$</td>
<td>10</td>
<td>1.3</td>
</tr>
<tr>
<td>$n_5$</td>
<td>18</td>
<td>0.72</td>
</tr>
</tbody>
</table>

For example in Table 3, there are 5 nodes in the cluster. We process the same program on each node and record the corresponding processing time. Then the capacity factors can be calculated easily. As Table 3 shows, node $n_4$ has the smallest capacity factor, which means it has the worst performance in the cluster.

By using the capacity factor, we modify the TD scheduler for heterogeneous environments. For each running job $j$, we denote the set of slots occupied by $j$ as $S_j$. For each slot $s$ in $S_j$, the node which $s$ belongs to is denoted as NodeOf $(s)$. Then, $j.occupied$ can be rewritten as

$$j.occupied = \sum_{s \in S_j} \text{NodeOf}(s).cf$$

(7)

Besides, we need to formalize the remaining workload of a node in a heterogeneous cluster. Then, Equation 6 is rewritten as

$$R_n = n.cf \times \sum_{j \in S_{adv} \cup S_{adv}} j.amt \times rt_n^j$$

(8)

With the simple modifications above, TD scheduler can be extended to heterogeneous environments easily.

6. **EXPERIMENTAL EVALUATION**

In this section, we evaluate the performance of TD scheduler. The MapReduce cluster for experiments is constituted by Hadoop (version 1.0.4) with 1 master and 50 slave nodes. Each node has a Core i3 2100 @ 3.1GHz CPU, 8GB memory, 500GB hard disk. The operating system is Red Hat Linux 6.1. There are 2 map task slots and 2 reduce task slots on each node, and the size of the input split is 64MB. The default value of the lower bound $L=0.7$, upper bound $H=1.3$. 
we use the following 4 types of MapReduce jobs to constitute the job queue for testing.

**Word count** is the basic MapReduce job. For each word \(w\), the map function output a pair of type \((w, 1)\). The reduce function sums the "1"s of a given word, and outputs a pair of type \((w, \text{counter})\).

**Inverted index** is used for building index for a large file. The map function parses the input file. For each word \(w\), it outputs a pair \((w, \text{position of } w)\). The reduce function merges the positions of a given word and outputs pairs of type \((w, \text{list(positions)})\).

**Distributed grep** is used for finding a give pattern from a large file. The map function outputs an intermediate result if it matches the pattern. The reduce function just copies the intermediates and outputs them.

**Distributed sort** is used for sorting large number of items in parallel, and it adopts continuous partition function. The map function just scans the items and sends them to the corresponding reducers based the partition function. The reduce function sorts the received items locally.

For each job \(j\), \(j.demand \in [15-40]\), and it is related to the job size (the split number of input file).

In order to imitate different real applications, we also generate 3 types of job queues. a) Small type means the size of each job in the queue is small (<100). b) Large type means each job size is large (>200). c) Mix type means the job size can be varied among [50 - 250]. The total input file size of the jobs in a queue spans [100 - 300] GB, the default value is 200GB.

We evaluation the performance of TD scheduler by comparing with the common FIFO scheduler and Fair scheduler. Firstly, we test the runtime of job queues, ratio of local task assignment and the occurrence number of hotspots. Then, we show the influence of the threshold. At last, we test the performance of TD scheduler in heterogeneous environments.

### 6.1 Result Analysis

**Figure 4. Runtime of Job Queues**

(a) Small Jobs

(b) Mix Jobs

(c) Large Jobs

**Figure 5. Ratio of Local Task Assignment**

(a) Small Jobs

(b) Mix Jobs

(c) Large Jobs

**Figure 6. The Influence of Threshold**

(a) Runtime vs. Lower Bound \(H=1.3\)

(b) Runtime vs. Upper Bound \(L=0.7\)
In Figure 4, we test the runtime of the job queues using different schedulers. For each type of job queue, the runtime time of our TD scheduler is significantly shorter than that of the others, which means TD scheduler can provide better throughput. Although the FIFO scheduler shows better performance than Fair scheduler (close to TD), it is not designed for multi users sharing a cluster. Besides, by comparing the 3 subfigures in Figure 4, more small jobs in a job queue leads to more time consumption.

In Figure 5, We test the network overhead through calculating the ratio of local task assignment. For the large type, Fair and FIFO maintains high ratio of local assignment (about 70%). With more small jobs in a queue, the local ratio decreases. For the small type, the ratio is reduced to nearly 35%. Whereas, TD scheduler keeps a high ratio of local assignment to reduce the network overhead, which is the main reason that TD can provide good throughput.

We also test the occurrence number of hotspots. The whole process is divided into 5 stages. We repeat the same process 3 times and record the average numbers of hotspots in each stage. The result is showed in Table 4. Comparing with the other 2 scheduler, TD scheduler can avoid majority of the hotspots. Note that there are still some hotspots in TD, which is because: a) Considering the network overhead, the nonlocal assignment is not allowed to be off-rack. b) The distribution of data may be not quite balanced. Hence in the final stage, the hotspots are unavoidable.

<table>
<thead>
<tr>
<th>scheduler</th>
<th>queue type</th>
<th>stage in processing (percent)</th>
<th>total</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td></td>
<td>1-20</td>
<td></td>
</tr>
<tr>
<td>Fair</td>
<td>small</td>
<td>25</td>
<td>121</td>
</tr>
<tr>
<td></td>
<td>mix</td>
<td>18</td>
<td>87.3</td>
</tr>
<tr>
<td></td>
<td>large</td>
<td>3</td>
<td>9</td>
</tr>
<tr>
<td>FIFO</td>
<td>small</td>
<td>23</td>
<td>108.7</td>
</tr>
<tr>
<td></td>
<td>mix</td>
<td>14</td>
<td>78.3</td>
</tr>
<tr>
<td></td>
<td>large</td>
<td>2</td>
<td>6</td>
</tr>
<tr>
<td>TD</td>
<td>small</td>
<td>0</td>
<td>2.4</td>
</tr>
<tr>
<td></td>
<td>mix</td>
<td>0</td>
<td>2</td>
</tr>
<tr>
<td></td>
<td>large</td>
<td>0</td>
<td>0.7</td>
</tr>
</tbody>
</table>

Table 4 Occurrence Number of Hotspots

Figure 6(a) describes the influence of lower bound. When $L>0.7$, with the increase of $L$, the runtime gets longer, which means that the throughput gets lower. This is mainly because higher $L$ leads to tighter constraint of jobs, so more forced assignments happen when slots requires tasks. When $L<0.7$, the constraint of jobs is loose enough, so the runtime gets stable. But too small $L$ may results in that some jobs are processed for quite a long time, which is not good for the user experience and the system flexibility. Hence the optimal value of the lower bound is 0.7. Similarly, we get the optimal value of the upper bound $H=1.3$.

6.2 Evaluation in a Heterogeneous Cluster

In order to build a heterogeneous cluster, we classify the 50 slave nodes into 3 groups and lock the CPU main frequency of the nodes in each group. The detail configuration is shown in Table 5.

<table>
<thead>
<tr>
<th>group ID</th>
<th>number of nodes</th>
<th>CPU frequency</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>16</td>
<td>2GHz</td>
</tr>
<tr>
<td>2</td>
<td>17</td>
<td>2.5GHz</td>
</tr>
<tr>
<td>3</td>
<td>17</td>
<td>3.1GHz</td>
</tr>
</tbody>
</table>

Figure 7 shows the runtime in a heterogeneous cluster. Comparing with the results in Figure 4, we can see that the runtime gets longer because we reduce the CPU frequencies of some nodes. TD scheduler still shows better performance than FIFO and Fair, and it shows stable performance even in a heterogeneous cluster.

Through the experiments above, we verify the effectiveness of TD scheduler and show that our proposed scheduler can improve the system throughput significantly.

7. CONCLUSIONS

This paper focuses on the system throughput problem for batch jobs in a shared MapReduce cluster, and proposes a novel task scheduling algorithm, called TD scheduler. First, according to the difference of the job parameters, we classify the jobs into 6 states and design the rules of state conversion, which can ensure the fairness of resource allocation and avoid the waste of computing resources. We design the detailed strategies for job selection and task selection. The strategies improve the ratio of local task assignment and avoid the occurrence of hotspots, which can significantly improve the system throughput. We also show that our TD scheduler can be applied in the heterogeneous MapReduce cluster through a small amount of modifications. At last, we verify the effectiveness and practicability of TD scheduler through plenty of experiments.
8. ACKNOWLEDGMENT
This work is supported by the National Basic Research 973 Program of China under Grant No.2012CB316201, the National Natural Science Foundation of China under Grant Nos. 61033007, 61003060, the Fundamental Research Funds for the Central Universities under Grant No. N100704001, the National Research Foundation for the Doctoral Program of Higher Education of China Grant No. 20120042110028 and the MOE-Intel Special Fund of Information Technology under Grant No. MOE-INTEL-2012-06.

9. REFERENCES
http://hadoop.apache.org/docs/r0.20.2/capacity_scheduling.html
http://hadoop.apache.org/docs/r0.20.2/fair_scheduling.html
JSSPP, Atlanta, USA, Apr 23, 2010

Authors
Xite Wang, born in 1987, Ph. D. student. His research area is big-data management.

Derong Shen, born in 1964, Ph. D., professor, Ph. D. supervisor. Her research area includes entity search and distributed computing.

Ge Yu, born in 1962, Ph. D., professor, Ph. D. supervisor. His research area includes database and big-data management.

Tiezheng Nie, born in 1980, Ph. D., associate professor. His research area includes data quality, data integration.

Yue Kou, born in 1980, Ph. D., associate professor. Her research area includes entity resolution, web data management.