Cluster Comput
DOI 10.1007/s10586-012-0236-5

A MapReduce task scheduling algorithm for deadline constraints

Zhuo Tang · Junqing Zhou · Kenli Li · Ruixuan Li

Received: 15 April 2012 / Accepted: 15 October 2012
© Springer Science+Business Media New York 2012

Abstract The current works about MapReduce task scheduling with deadline constraints neither take the differences of Map and Reduce task, nor the cluster's heterogeneity into account. This paper proposes an extensional MapReduce Task Scheduling algorithm for Deadline constraints in Hadoop platform: MTSD. It allows user specify a job's deadline and tries to make the job be finished before the deadline. Through measuring the node's computing capacity, a node classification algorithm is proposed in MTSD. This algorithm classifies the nodes into several levels in heterogeneous clusters. Under this algorithm, we firstly illuminate a novel data distribution model which distributes data according to the node's capacity level respectively. The experiments show that the node classification algorithm can improved data locality observably to compare with default scheduler and it also can improve other scheduler's locality. Secondly, we calculate the task's average completion time which is based on the node level. It improves the precision of task's remaining time evaluation. Finally, MTSD provides a mechanism to decide which job's task should be scheduled by calculating the Map and Reduce task slot requirements.

Z. Tang (✉) · J. Zhou · K. Li
School of Information Science and Engineering, Hunan University, Changsha 410082 Hunan, China
e-mail: ztang@hnu.edu.cn

J. Zhou
e-mail: zjq@hnu.edu.cn

K. Li
e-mail: lkl@hnu.edu.cn

R. Li
School of Computer Science and Technology, Huazhong University of Science and Technology, Wuhan 430074, Hubei, China
e-mail: rxli@hust.edu.cn

Keywords MapReduce · Scheduling algorithm · Data locality · Deadline constraints · Hadoop

1 Introduction

Nowadays, requirements for massive data processing are increasing, such as machine learning [1], scientific analysis [2], astrophysics [3] and bioinformatics [4]. There are several programming model for processing massive data, such as Dryad [5], Scatter-Gather-Merge [6], and MapReduce [7]. MapReduce is a distributed programming model for expressing distributed computation on massive amounts of data and an execution framework for large-scale data processing on clusters of commodity servers, it has been implemented in multi environments, such as Mar [8], Phoenix [9], and Hadoop [10]. In [11], the authors have used the MapReduce along with boosting model to work out data abundance. For the model can be realized using the low-cost computers, MapReduce framework is becoming more and more popular in various applications. One of the most important advantages of MapReduce is its convenience, such that, programmers can process massive data without knowing the details of distributed implementation, and users can process large scale of data by only providing the Map and Reduce interface. The Map and Reduce stage is strict in the original MapReduce model, but there are some works try to break the barrier [12, 13].

The initial MapReduce model was designed for off-line data processing. However, it is now popularly applied in heterogeneous, sharing and multi-user environments. The research of the MapReduce scheduling algorithm mainly in four areas: (1) the data locality of the MapReduce tasks. It is the effect of the data distribution to task scheduling; (2) fault-tolerant scheduling and expectation execution

Published online: 12 December 2012

time in a heterogeneous environment; (3) resource sharing: for the hadoop cluster, how to share the computing resources through scheduling the user groups; (4) resource-aware scheduling algorithm. It is based on the status of the cluster resources , such as memory, disk IO, network, and other factors; (5) real-time scheduling. It is the study for the MapReduce real-time scheduling model. Nowadays, the MapReduce scheduling algorithms mainly include FIFO (First Input First Output), LATE (Longest Approximate Time to End) [14], FairScheduler [15] and CapacityScheduler [16]. Basic features such as data locality, user priority, fault-tolerant and faiHrness are all considered by these algorithms. Moreover, few algorithms have considered the user's job deadline constraints, such as in the elastic cloud computing environment or online service system [17]. Through the job deadline, we can build a model to advance the veracity of the task remaining time estimating in the heterogeneous environment, make the use jobs can be finished before the deadline furthest.

To meet the users' job deadline requirement in the cloud environment we propose the MapReduce Task Scheduler for Deadline (MTSD) algorithm. The MTSD algorithm takes the data locality and cluster heterogeneity into account. The data locality is the key factor that affects the efficiency of MapReduce jobs' running. The data locality means that the task's operation code and the task's input data are on the same computing node or the same rack. Of course, the efficiency when the code and data are on the same node is higher than on the same rack. If the code and data are on the same node, it would avoid the data transmission in the network and greatly reduce the delay. Therefore, in the large-scale data processing applications, shifting the code would be "cheaper" than moving data.In this paper, in order to meet the time constraints of the job and further improve the data locality, the MapReduce Task Scheduler for Deadline (MTSD) algorithm is proposed, which based on the computing power of meet time constraints in the heterogeneous environments .

The contributions of this paper are as follows: (1) we introduce A node classification algorithm to improve the Map task's data locality. (2) We present a novel remaining task execution time model which is based on node classification algorithm.

The rest of this paper is organized as follows. Section 2 reviews the related works. In Sect. 3, we propose the MTSD algorithm to meet the deadline constraints, and node classification algorithm to improve the Map task's locality. In Sect. 4, we present the experiment results of data locality, job scheduling and task remaining estimate. We conclude the paper and future works in Sect. 5.

2 Related work

At present, the researches on MapReduce scheduling algorithms focus on data locality, sharing, fairness and fault-tolerant ability. Dynamic Proportional Scheduler [18] provides more job sharing and prioritization capability in scheduling and also results in increasing share of cluster resources and more differentiation in service levels of different jobs. Data locality is a key performance factor of task's completion time in Hadoop. Matei Zaharia et al. propose the delay scheduling algorithm [19] to address the conflict between data locality and fairness. However, the method takes fairness withered as the cost and it doesn't fit for the jobs which have large size or few slots per node. The data locality in MapReduce means that a Map task is executed on the node which contains its input data. In [20], the authors improve data locality through building a relationship between application and nodes to place data reasonable. Matei Zaharia et al. [19] propose the delay scheduling algorithm to address the conflict between locality and fairness. Xiaohong Zhang et al. propose the NKS algorithm [21] to improve the data locality of map tasks. However, it is based on the homogeneous environment.

There are some works on MapReduce job scheduling algorithm take deadline constraints into account. Time estimation and optimization for Hadoop jobs has been explored by [22, 23]. In [22], the authors focus on minimizing the total completion time of a set of MapReduce jobs. In [23], it estimates the progress of queries that run as MapReduce DAGs. The works in [24, 25] propose solutions for MapReduce job scheduling on deadline constraints problem. Kamal Kc et al. [24] propose a task execution time model by evaluating the tasks' data processing unit time and data transferring unit time, and then it uses the model to compute how many Map and Reduce tasks should be scheduled to satisfy the deadline constraints. Polo et al. [25] propose a task scheduler to predict the performance of concurrent MapReduce jobs dynamically and adjust resource allocation for the jobs. However, it didn't consider the differences between the Map and Reduce tasks. Both of the Map and Reduce task's execution time are uncorrelated, so it's not accurate to compute average task execution time by taking Map and Reduce tasks together. In [26], authors present a formal model for capturing real-time MapReduce applications and the Hadoop platform. The above scheduling algorithms are based on homogeneous cluster. Heterogeneous clusters [27] consist of kinds of nodes with different performance characteristics in computing power, memory capacity and disk speed. So the homogeneous scheduling algorithms can't deal with deadline constraints efficiently in heterogeneous environment.

In this paper, we propose MTSD algorithm for job's deadline constraints in heterogeneous environment. We take the cluster's heterogeneity into account by proposing a node

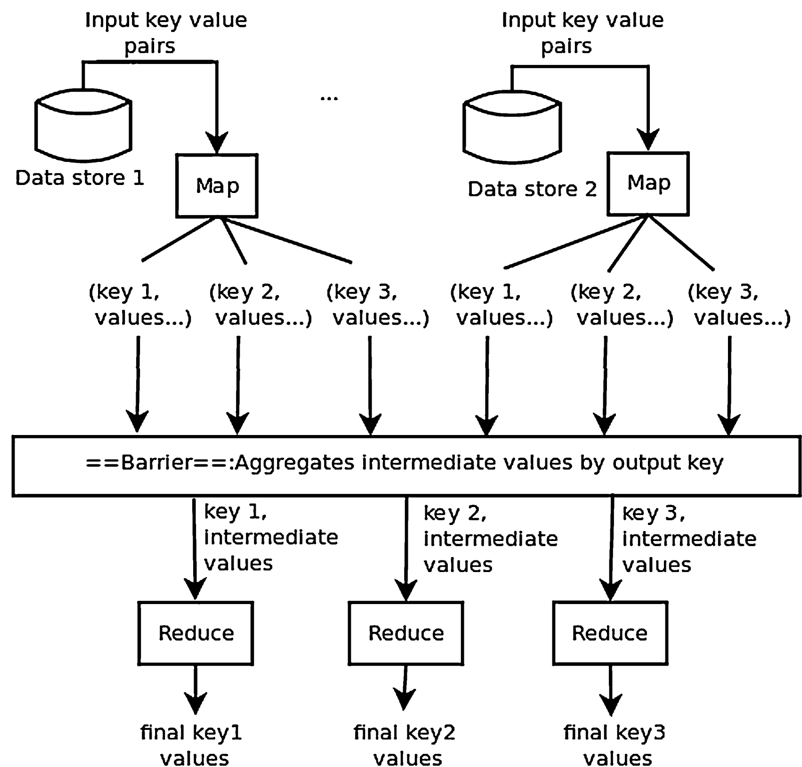


Fig. 1 The MapReduce computing framework

classification algorithm. This method improves the Map tasks' data locality, and the experiments show that the node classification algorithm can improved data locality about 57 % to compare with default scheduler and it also can improve other scheduler's locality. Because of the uncorrelated of Map and Reduce task, we separate the Map and Reduce scheduling into two phases, and MTSD provides a mechanism to decide which job's task should be scheduled by calculating the Map and Reduce task slot requirements. The experiment results show that MTSD algorithm can meet the deadline constraints requirement.

3 MTSD: deadline constraints based MapReduce scheduling algorithm

3.1 MapReduce computing framework and data locality

As a distributed computing framework on commercial computer, one of the MapReduce's most significant advantages is that it provides an abstraction that hides many system-level details from programmer. It processes data by dividing the progress into two phases: Map and Reduce. Each Map function takes a split file as its input data, which locates in the distributed file system and contains the key-value data. The split file can be co-location with the Map function or not. If the split file and the Map function don't in the same node, then the system will transfer the split file to the Map function. This procedure will delay the execution of Map task. Figure 1 shows that the Map function is applied to each input key-value pair and generates an arbitrary number of intermediate key-value pairs. The procedure of Map task includes [28]:

- Read: Reading the input split and creating the key-value pairs.
- Map: Executing the user-provided map function.
- Merge and Write: Collecting the map output into a buffer and partitioning. Writing the buffer to disk, as a spill file. Merging the file spills into a single map output file. Merging might be performed in multiple rounds.

And the procedure of Reduce task includes:

- Shuffle: Copying the map output from the mapper nodes to a reducer's node and decompressing, if needed. Partial merging may also occur during this phase.
- Merge: Merging the sorted fragments from the different mappers to form the input to the reduce function.
- Reduce: Executing the user-provided reduce function.
- Write: Writing the output to HDFS (Hadoop Distributed File System).

The Reduce function is applied to all values that associated with the same intermediate key and generates output key-value pairs as the final result. Hadoop is an open source implementation of MapReduce.

In the MapReduce framework, Map or Reduce codes can be moved among the cluster nodes and the data can be trans-

ferred from a node to another. If the code and data on the same node, we call this "data locality". The cost of migrating code is extremely lower than migrating data. So the ideal situation is to move the code, not data. For the sake of this purpose, it needs a reasonable data distribution strategy. The default data distribution strategy in Hadoop is random. In this paper, we show a new data distribution model to improve data locality.

3.2 Node classification algorithm

As a data-intensive computing framework, most of MapReduce's jobs are toward the massive data processing. And the task's completion time is related to the factors of computer's CPU, disk, memory and etc.

In this paper, we use the comprehensive criterion to represent the data processing speed. This criterion should be the comprehensive processing capacity facing all memory intensity, IO intensity and CPU intensity jobs. In a heterogeneous environment, cluster usually contains nodes with different processing capacity, which means the speed of the node processes data. We classify the nodes according to this processing capacity. There are two purposes of node classification with their processing capacity: one is to optimize the data distribution in order to improve the data locality; the other is to improve the evaluation accuracy of the task remaining time in heterogeneous environment.

Definition 1 Assure that there are K levels of nodes in cluster, L_p $(1 \le p \le K)$ means the level factor of the p-th level, and let L_p as:

$$L_p = \begin{cases} L_{p-1} * \delta & 1 < p \le K \\ 1 & p = 1 \end{cases} \tag{1}$$

where δ is a constant number and is greater than 1, which is named classification factor.

In order to get the differences between nodes with their capacities, we divide the nodes to different levels. In our cluster, we take the slowest data processing speed nodes as the first level and the level factor is 1. And the L_p $(1 \le p \le K)$ represent the p-th level of servers' processing capacity, which is recursive calculated by multiplying L_{p-1} and the classification factor δ. So we know that L_p is larger than L_{p-1}. And the nodes that belong to L_p has a better capacity than the nodes that belong to L_{p-1}. According to the definition of L_p, we can separate the heterogeneous environment nodes to different levels simply.

We quantize the node's computing capacity simply by running a group of specific tasks. On a given cluster, we set each node with one Map slot and one Reduce slot. And all the map tasks are feed by the same input data. Then the benchmark jobs run on the cluster. We record the completion time of each task on where it ran. We set δ as a constant value larger than 1. Pseudo codes for node classification algorithm are shown in Algorithm 1.

Note that different benchmark jobs will make a different classification result. For example, a node may be in level i on job1 while be in level j on job2. To resolve this problem, we test a series of benchmark jobs and use a statistics method to decide which level the node will be belong to.

The node classification algorithm enables us to decide the nodes' level by their computing capacity. The data distribution strategy is that the size of each node's data is in proportion to the node's level. For example, there are three levels of cluster nodes L1, L2 and L3; and the size of data is 60 G. Their computation ability is 1, 2, 3 grade, so the data distribution on nodes is 10 G, 20 G and 30 G respectively. The data in the same type is distributed by random distribution principle. In the same class of nodes, it selects a node to store data randomly. The node classification algorithm is shown in Algorithm 1.

Algorithm 1 Node classification algorithm

```
N = {1, 2, ..., n}        /*node collection*/
CLASS[] = NULL            /*level collection*/
level = 1                 /*the level indicator*/
for each node ∈ N
   compute the benchmark task on node
   T[i].time = get the task's completion time on node
   T[i].node = node
end for
sort T[] by time in descending order
node = T[1].node
add node to CLASS[level]
//classify the nodes to different levels base the slowest node.
for j from 2 to T.length-1
   set a = task's completion time on node /T[j].time
   set level = 1
   j = δ
   while (j < a)
   {
      j = j * δ
      level++;
   }
   add T[j].node to CLASS[level]
add T[j].node to CLASS[level]end for
return CLASS
```

From the Algorithm 1, we can see that the algorithm computes a same task for all the n nodes and get their execution time. We sort the nodes by their completion time in descending order. We named the ordered nodes with node 1, node 2, ... and node n. Node 1 has the least completion time.

As the node queue is descending order of computing power, with the first as a benchmark, we classify the nodes to different levels base the ratio between the completion times of the nodes. The node 1 is in the level 1, and it is the lowest node. Because the computing power each levels is geometric increasing, the incremental change is δ. Hence, we use the complete time of the node i to divide the first node's time, and decide its level through the power of the value δ to the divide result.

3.3 The remaining task execution time model

In this section, we construct a model to calculate the remaining time of all running Map or Reduce tasks. In order to get the slot requirement of running job, we must estimate the remaining task execution time. Evaluation the remaining time of a task is mainly on the average competition time that the same type of tasks on the same level of nodes.

Definition 2 $J = (M, R, A, D)$ means a MapReduce job. M, R, A and D denotes Map task set, Reduce task set, job arrived time, and job deadline constraints respectively.

$CM(J, p)$ denotes the completed job J's Map task set which run on the p-th level; and $CR(J, p)$ means the already completed Reduce task set which ran on the p-th level; TM_m denotes the task M_m's ($M_m \in CM(J, p)$) completion time; TR_r denotes the task R_r's ($R_r \in CR(J, p)$) completion time. The average completion time of job J's Map tasks, which run on the p-th level should be calculated by all the Map jobs that run on the p-th level nodes's completion time divided the completed job J's Map task set which run on the p-th level. So,

$$\text{Average_TM}(J, p) = \frac{\sum_{M_m \in CM(J,p)} TM_m}{|CM(J, p)|} \tag{2}$$

And the average completion time of job J's Reduce tasks', which run on the p-th level should be calculated by all the Reduce jobs that run on the p-th level nodes's completion time divided the completed job J's Reduce task set which run on the p-th level. So,

$$\text{Average_TR}(J, p) = \frac{\sum_{R_r \in CR(J,p)} TR_r}{|CR(J, p)|} \tag{3}$$

We use $UM(J)$ and $UR(J)$ to denote waiting task set of the job J's Map and Reduce task type respectively. $MM_m(J, p)$ denotes the completion time of job J's Map task which runs on the node in p-th level. So the completion time of a running task equal to the running time that have spent and the remaining execution time. That is:

$$MM_m(J, p) = RTM_m + CTM_m \tag{4}$$

where CTM_m means the m-th Map task's running time that have spent, RTM_m means the m-th Map task's remaining execution time. So the m-th Map task's remaining execution time of job J is:

$$RTM_m = MM_m(J, p) - CTM_m \tag{5}$$

The completion time of a certain job's tasks, which run on the nodes belonging to the same capacity level, will tend to be the same. We know that $MMm(J, p)$ is approximately equal to Average_TM(J, p). So Eq. (5) can be revised as:

$$RTM_m = \text{Average_TM}(J, p) - CTM_m \tag{6}$$

In the same way, we can get the running Reduce task's remaining execution time:

$$RTR_r = \text{Average_TR}(J, p) - CTR_r \tag{7}$$

where CTR_r denotes the running time that have spent. Because the value of RTM_m or RTR_r which is based on the tasks' mean time, so we can not get the first Map or Reduce task's remaining time value. In the experiment we simply treat it as an infinity value in the beginning.

Now we can calculate the sum of the remaining Map and Reduce task's execution time of job J which running on the p-th level nodes:

$$SM(J, p) = \sum_{1 \le i \le m} RTM_m \tag{8}$$

$$SR(J, p) = \sum_{1 \le i \le r} RTR_r \tag{9}$$

3.4 Deadline constraints based task scheduling algorithm

3.4.1 Map and Reduce's two stage scheduling

In MapReduce, the job's execution progress includes Map and Reduce stage. So, the job's completion time contains Map execution time and Reduce execution time.

In view of the differences between Map and Reduce's code, we divide the scheduling progress into two stages, namely Map stage and Reduce stage. Previous researches usually simplify the Map and Reduce as the same type of scheduling problem. It may do simplify the problem, but it is improper for the reason that the execution of Map and Reduce's code is different. In the aspect of the task's scheduling time prediction, the execution time of Map and Reduce is not correlative; their execution time depends on the input data and function of their own. Therefore, in this paper the scheduling algorithm sets two deadlines: map-deadline and reduce-deadline. And reduce-deadline is just the users' job deadline.

In order to get map-deadline, we need to know the Map task's time proportion on the task's execution time. In a cluster with limited resources, Map slot and Reduce slot number is decided. For an arbitrary submitted job with deadline constraints, the scheduler has to schedule reasonable with the remaining resources in order to assure that all jobs can be finished before the deadline constraints.

Definition 3 P_m and P_r are the proportion of Map and Reduce task's execution time respectively, and $P_m + P_r = 1$.

This paper uses empirical data to quantize the value of P_m and P_r. We use a data sample from user's input data and run the code on the data, and then we get the map and reduce task's execution time T_m and T_r. So:

$$P_m = \frac{T_m}{T_m + T_r} \tag{10}$$

and

$$P_r = \frac{T_r}{T_m + T_r} \tag{11}$$

Now we can calculate the map-deadline by P_m:

$$D_m = A + (D - A) * P_m \tag{12}$$

where A and D means the job's arrived time and deadline respectively.

According to map-deadline, we can acquire the current map task's slot number it needs; and with reduce-deadline, we can get the current reduce task's slot number it needs.

We estimate the time needed for the remaining task on the lowest level node. By this way, we can find the emergency degree of current job and the minimum Map slot requirement of job J can be computed as follows:

$$\delta_J^m = \frac{\sum_{1 \le p \le K} SM(J, p) * L_p^p + UM(J) * MM(J, 1)}{|D_m - CurrentTime|} \tag{13}$$

Similarly, the minimum Reduce slot number requirement of job J is:

$$\delta_J^r = \frac{\sum_{1 \le p \le K} SR(J, p) * L_p^p + UR(J) * MR(J, 1)}{|D - CurrentTime|} \tag{14}$$

3.4.2 Deadline constraints based task scheduling algorithm

The scheduling strategy of MSTD is based on δ_J^m and δ_J^r. The minimum Map and Reduce slot number required of job J can be denoted as δ_J^m and δ_J^r respectively. The symbol δ_J^m reflects that δ_J^m Map tasks should be scheduled at present in order to meet job J's map-deadline, as well as to meet the reduce-deadline (job deadline) δ_J^r Reduce tasks should be scheduled. In the scheduling process, we take δ_J^m and δ_J^r as the basic criteria of priority allocation.

But there are some special cases should be considered: (1) At the beginning of the job be submitted, there is no data available, so the scheduler can't estimate the required slots or the completion time of tasks. In this case, the job's precedence is over than the others. (2) In some scenarios, jobs may have already missed their deadline. The strategy we use is the same as the previous case: set such jobs' tasks with the highest priority. Algorithm 2 proposes the MTSD scheduling method. The input parameter t which containing the free map slots and reduce slots, represents a request to ask for waiting tasks. And the return value is a collection T which contains the tasks to be assigned to task tracker.

The MapReduce scheduling flows in Hadoop is as follows: (1) The scheduling maintains a job queue; (2) The work node, which is called TaskTracker, asks the scheduler for tasks periodicity; (3) When a request arriving, the scheduler decides which task to be assigned to the TaskTracker according the scheduling algorithm. In Algorithm 2, we firstly compute δ_J^m and δ_J^r for each job in queue and resort the jobs' order (see lines 6–8). Secondly, on lines 9–21 and lines 22–34, we schedule the waiting Map and Reduce tasks respectively. Finally, return the task collection T to the TaskTrackter and the TaskTracker will execute the tasks.

Table 1 The hardware configuration

Level	CPU	Memory	Amount
I	4-core, 3.07 GHz	4 G	8
II	4-core, 2.7s GHz	4 G	10

4 Evaluation

In this section, we design three aspects of experiments to evaluate MSTD, including: (1) Data locality: We evaluate the MTSD scheduler's data locality. Firstly, we compare the data locality improvement with the default scheduler: FIFO; then, we also evaluate the effect of node classification algorithm to other scheduler: FairScheduler. (2) We estimate the job scheduling behavior of MTSD by submitting a series of jobs with different deadlines. The results show that MTSD can perform a good behavior to the deadline jobs. (3) In the third section, we present the relative error of task remaining time estimation. We have carried out a comprehensive set of experiments in order to evaluate the effectiveness of MTSD scheduling algorithm. We use three typical MapReduce programs as the running example, WordCount, Join and Pi. And the WordCount is a program to count each word appears in the text file. The Join program can join the two tables on a given column. Pi is using the Monte Carlo method to compute π. We use RandomWriter to generate experiment data. These programs are all released with the Hadoop library with its source codes.

In the experiments, we generate job input files by a random program, and use two node levels and let $\delta = 1.2$. Each node has four map slots and four reduce slots. Table 1 shows the hardware configuration in our experiments.

4.1 Data locality

Data locality is a key performance of Hadoop MapReduce jobs. In Sect. 3.2, we propose a node classification algo-

Algorithm 2 MTSD scheduling algorithm

```
1.  Collection assignTask(TaskTracker t)
2.  M = t.freeMapSlots                          /* M is the map slot collection */
3.  R = t.freeReduceSlots                       /* R is the reduce slot collection */
4.  Q = ∅                                       /*job queue */
5.  T = ∅                                       /*task collection, to be scheduled*/
6.  compute the slot requirement for jobs in queue Q
7.  update jobs' priority in queue Q            /* according δ^m_J, δ^r_J and time * /
8.  resort the jobs in queue Q                  /* sort jobs in descending order */
9.  for each job ∈ Q do
10.   count = 0
11.   if count < job.MapSlotRequirementCount
12.     if job exists waiting map task and M ≠ ∅
13.       Task t = job.obtainMapTask()
14.       T = T ∪ {t}
15.       remove a Map slot in M
16.       count++
17.     else break
18.   else break
19. for each job ∈ Q do
20.   count = 0
21.   if count < job.ReduceSlotRequirementCount
22.     if job exists waiting map task and R ≠ ∅
23.       Task t = job.obtainReduceTask()
24.       T = T ∪ {t}
25.       remove a Reduce slot in R
26.       count++
27.     else break
28.   else break
29. return T
```

rithm. So there are two ways of data distribution in our evaluation. The first one is according the classification algorithm and the job input data is distributed in proportion with the node computing capacity. The second one is distributing job input data in random, which is the default way of Hadoop.

We run WordCount, Join and TeraSort program to measure the proportion of the data locality. We compare the jobs' data locality performance with FIFO and FairScheduler. The FairScheduler uses a delay scheduling strategy, which make a great improvement on data locality. The delay scheduling is included in the FairScheduler since the version of Hadoop-0.21.So, we apply the delay scheduling strategy in our algorithm (called MTSD+Delay) to compare with native MTSD. The experiments are divided into two groups: The MTSD curve is with the node classification algorithm and the FIFO is a group experiment without node classification algorithm which is traditional algorithm. The results are shown in Fig. 2.

In the Fig. 2(a), (b) and (c), we show the locality on FIFO and MTSD. In the FIFO experiment,we find that the task runs in the nodes of level II is more than the task runs in the level I on average. But the locality of level II nodes is lower than the average locality. In Fig. 3(a), (b) and (c), we evaluate the locality in FairScheduler and MTSD with delay scheduling. There are two scheduling modes in FairScheduler, in our experiment we use the FIFO mode. The purpose of this evaluation is to observe that if the data distribution strategy we use in MTSD can improve the scheduler's locality. From Fig. 3, we can see that the MTSD with delay scheduling improve the data locality 10.5 %, 9.28 %, and 28.84 % while comparing with FairScheduler on average, in the running job of WordCount, Join and Tersort respectively.

In Fig. 4, we show the experiments' completion time in Fig. 3. From **this result**, we can see that the MTSD with delay scheduling reduce the jobs' completion time 12.3 %, 20.8 %, and 11.3 % while comparing with FairScheduler on average, in the running job of WordCount, Join and Tersort respectively.

It's clear that the data distribution strategy we use in MTSD improves the scheduler's locality. MTSD algorithm takes the heterogeneous environment into account, and proposes node classification algorithm to assess different

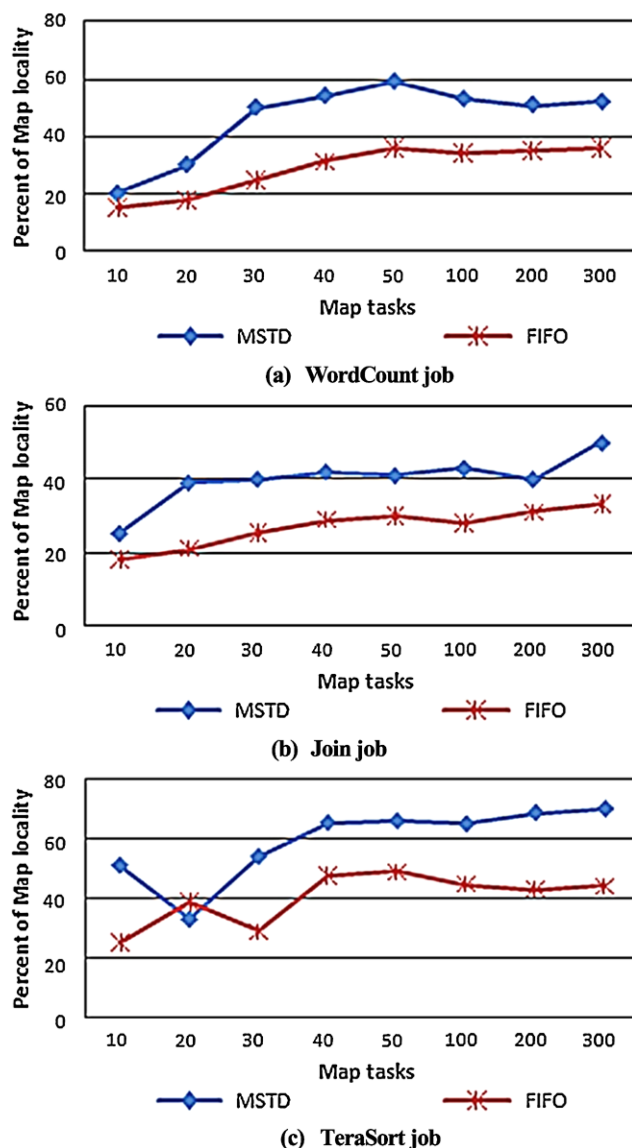


Fig. 2 The Map tasks' data locality

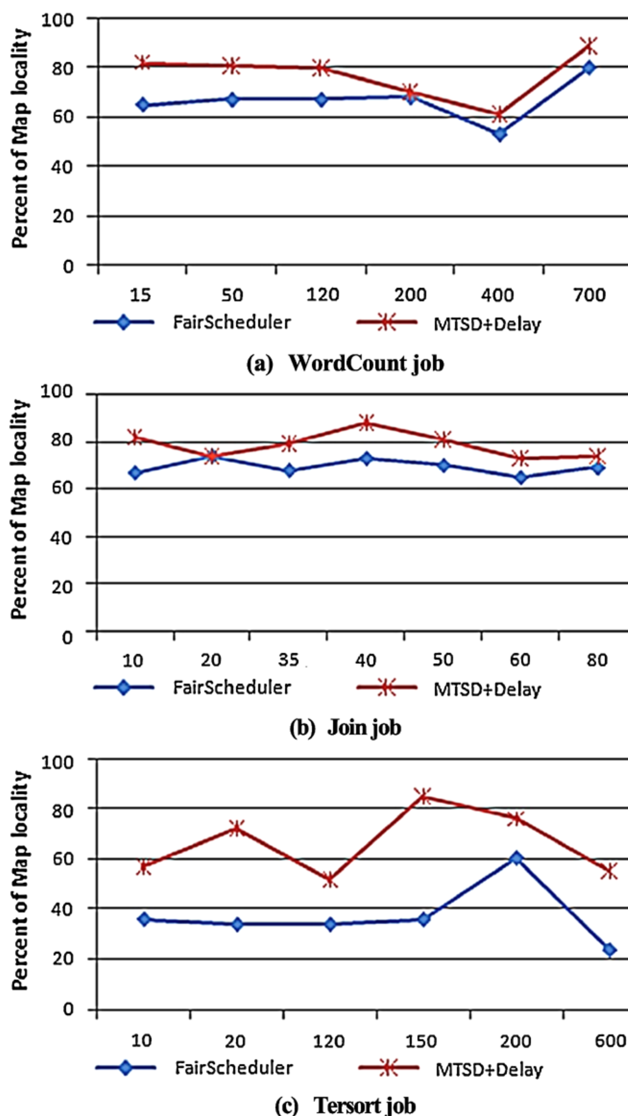


Fig. 3 FairScheduler vs MTSD+Delay

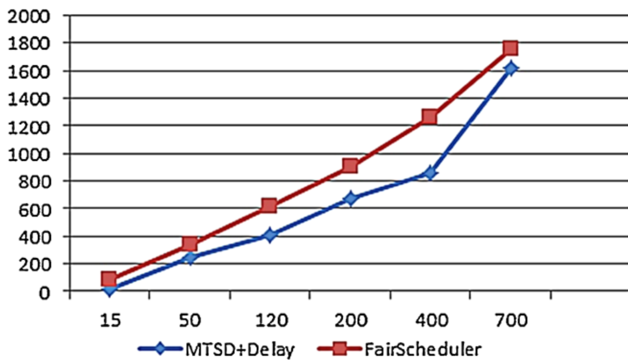
nodes with their capacity fairly. After processing nodes with MTSD, data distribution is more reasonable, so we can achieve a better data locality. The proportion of data locality can improve about 57 % when compare with the default FIFO scheduling.

4.2 Job scheduling

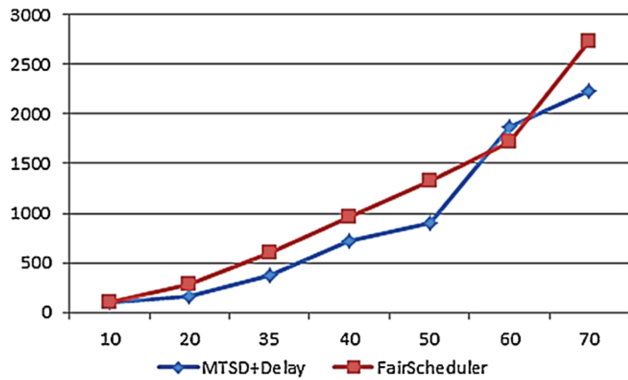
In order to evaluate the performance of MTSD algorithm, we use three jobs: J1, J2 and J3 stand for Gridmix, Join and WordCount respectively. The three job's assignments respectively in three time points: S1, S2, S3. Each job has its deadline. In Fig. 5(a), (b) and (c), the symbols M1, M2 and M3 means three jobs' map-deadline; R1, R2 and R3 means the three jobs' reduce-deadline. The horizontal indicates the time, and the unit is second.

At the time S1, in Fig. 5(a), only J1 is running, and the current free slot number is 60, MTSD algorithm will assign all slots to J1 for map task. At S2 moment, the job J2 arrives, $M2-S2 < M1-S1$, which means that J2 is more urgent than J1, so J2 will have a higher priority than J1. However, at S3, the user submits J3, and $M3-S3 < M2-S2$, obviously, J3 has a higher priority than J2, so j3 is given privileged access of slots. From S3 to M3, the system assign 50 slots to J3, the rest of the 10 slots assign to J2. When J3, J2 finished, MTSD will assigned most of the slots to j1 in order to assure that j1 can finish the task before m1. The reduce-deadline scheduling is similar to map-deadline scheduling. From Fig. 3 we can see that the closer the task to deadline, the higher priority the job gets.

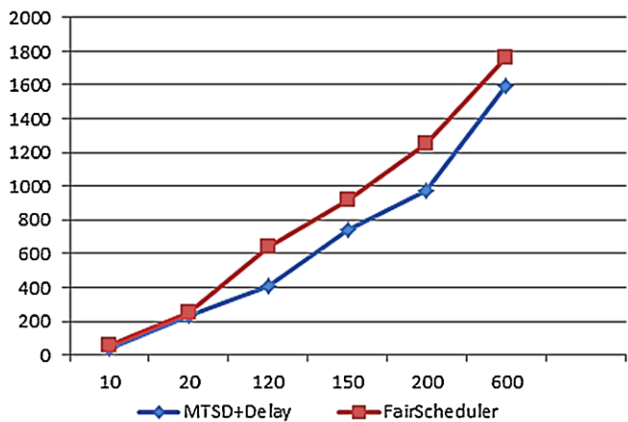
From Fig. 5, MTSD scheduling algorithm can satisfy the job's Deadline constraints, especially in multi-jobs environment. MTSD schedules the tasks with their urgent level. But,



(a) WordCount job

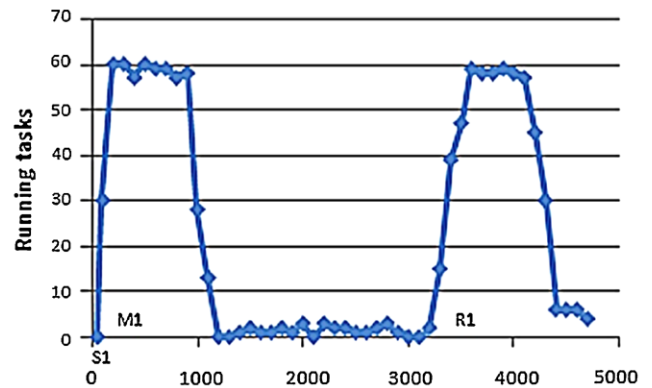


(b) Join job

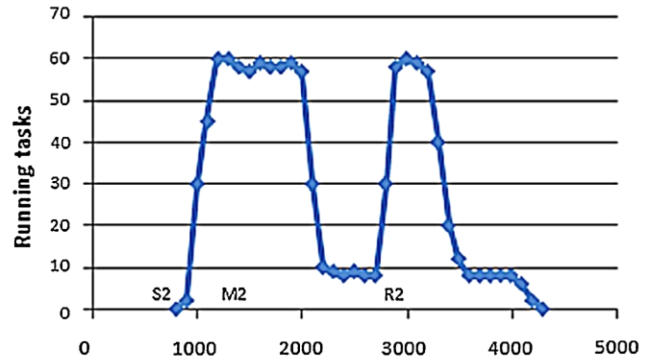


(c) TeraSort job

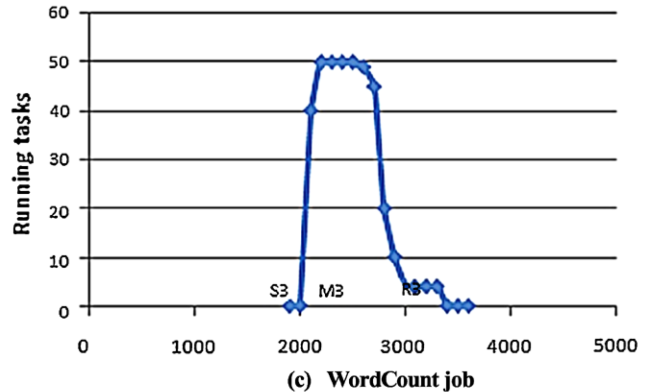
Fig. 4 The completion time comparing: FairScheduler vs MTSD+Delay



(a) TerSort job



(b) Join job



(c) WordCount job

Fig. 5 The progress of job scheduling

in some extreme cases, scheduler can't meet requirements. And if the running tasks' deadlines don't set reasonable, in some cases all the work can't finish their tasks before deadline.

There is a problem in our implementation is that the Reduce task scheduling issue. In Hadoop, the Reduce task includes three stages: shuffle, sort and reduce. And the shuffle stage is pulling the data from every Map node. There is no data locality for Reduce task. And the shuffle stage will not be finished until all of the Map tasks are completed. Thus, in our implementation we schedule the Reduce task as soon so possible.

4.3 Task remaining time estimation

In the formulate (7), we show an alternative way to estimate the remaining time of a task. In the following experiments we measure the corrective of this method and show the results. The relative error can be calculated as formulate (15). We measure two types of job: IO intensive and CPU intensive. These are WordCount, Pi, and TeraSort. And the WordCount and TeraSort are the IO intensive job, while the Pi is CPU intensive. In the Fig. 6(a) and Fig. 6(b), we find that the relative error is application aware. The average relative

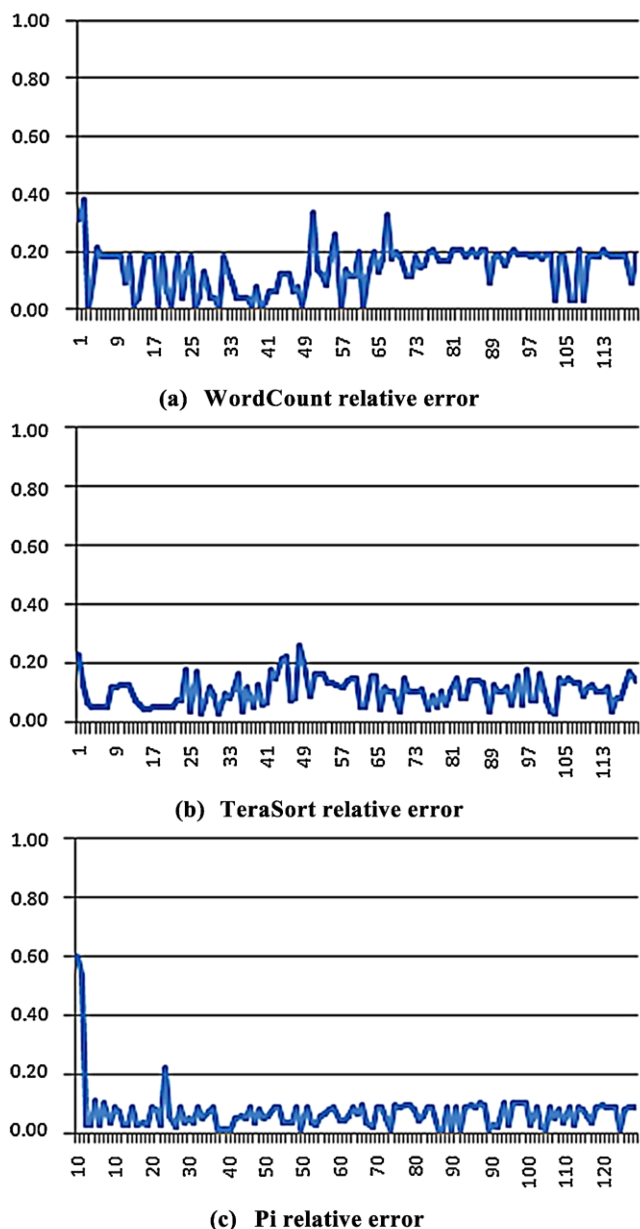


Fig. 6 The relative error evaluation: to estimate accuracy of time prediction method in MTSD

error is 0.14, 0.11 and 0.07 for WordCount, Tersort, and Pi respectively. The relative error of Pi is smaller than the else of two. Because Pi is a CPU-intensive application, and its Map tasks do not have input data, so it avoid the network delay compare with others.

$$relative_error = \frac{|estimate_time - real_time|}{real_time} \tag{15}$$

In the testing of WordCount and TeraSort, some of the tasks' relative errors are greater than 0.2, one of the reason is the locality problem, that is the Map task should read remote file and the actual completion time for that task will be larger than average.

We predict the task's remaining time by Eqs. (7) and (8). But it is based on finished tasks' mean time. So it can not estimate the tasks' remaining time at the beginning of the job submitted. To estimate the first task's completion time, we take a sample data from the job's input data and run only a map/reduce task for this job on this sample data [29]. We use this task's completion time as the initial value of the task's mean time. If there is one or more task completed, the other tasks' remaining can be estimated through (7) or (8). From the Fig. 5 we can see that, in the beginning the tasks' relative error is greater than later.

5 Conclusion and future work

User constraints such as deadlines are important requirements which are not considered by existing cloud-based data processing environments such as Hadoop. The MTSD proposed in this paper focuses on user's deadline constraints problem. In this algorithm, a node classification algorithm is proposed to divide the nodes into different type with their computation ability. With the classification results, we get the data distribution principle for the input data. We do two kinds of comparing work: the MTSD vs FIFO, and native MTSD vs MTSD with delay scheduling. And the experiments show that the data distribute strategy can improve both the MTSD and FairScheduler's data locality. The experiments show that most tasks can meet the deadline constraints. We propose two stages scheduling to accurate the scheduling progress, and improve the system's schedule performance. The MTSD is implemented in version of Hadoop-0.21.

The future work of this paper is to embed the node classification algorithm into the Hadoop distributed file system. And it also needs to find a good solution to solve the Reduce task scheduling problem.

Acknowledgements This work is supported by the National Natural Science Foundation of China (61103047, 61173170), National Post doctor Science Foundation of China (20100480936).

References

1. Chu, C.T., Kim, S.K., Lin, Y.A., Yu, Y.Y., Bradski, G., Ng, A.Y., Olukotun, K.: Map-reduce for machine learning on multicore. http://www.cs.standford.edu/peop;e/ang//papers/nips06-mapreducemulticoure.pdf (2006). Accessed 1 March 2012
2. Ekanayake, J., Pallickara, S., Fox, G.: MapReduce for data intensive scientific analyses. In: Proceedings of the 2008 IEEE Fourth International Conference on eScience, pp. 277–284 (2008). doi:10.1109/eScience.2008.59
3. Mackey, G., Sehrish, S., Bent, J., Lopez, J., Habib, S., Wang, J.: Introducing map-reduce to high end computing. In: Proceedings of the 2008 3rd Patascale Data Storage Workshop, pp. 1–6 (2008). doi:10.1109/PDSW.2008.4811889

4. Matsunaga, A., Tsugawa, M., Fortes, J.: CloudBLAST: combining MapReduce and virtualization on distributed resources for bioinformatics applications. In: Proceedings of 4th IEEE International Conference on eScience, pp. 222–229 (2008). doi:10.1109/eScience.2008.62
5. Isard, M., Budiu, M., Yu, Y., Birrell, A., Fetterly, D.: Dryad: distributed data-parallel programs from sequential building blocks. In: Proceedings of the 2nd ACM SIGOPS/EuroSys European Conference on Computer Systems, vol. 41, pp. 59–72 (2007). doi:10.1145/1272996.1273005
6. Han, H., Jung, H., Eom, H., Yeom, H.Y.: Scatter-Gather-Merge: an efficient star-join query processing algorithm for data-parallel frameworks. Clust. Comput. **14**(2), 183–197 (2010). doi:10.1007/s10586-010-01445
7. Dean, J., Ghemawat, S.: MapReduce: simplified data processing on large clusters. Commun. ACM **51**(1), 107–113 (2008). doi:10.1145/1327452.1327492
8. He, B., Luo, Q., Govindaraju, N.K.: Mars: accelerating MapReduce with graphics processors. IEEE Trans. Parallel Distrib. Syst. **22**(4), 608–620 (2011). doi:10.1109/TPDS.2010.158
9. Ranger, C., Raghuraman, R., Penmetsa, A., Bradski, G., Kozyrakis, C.: Evaluating mapreduce for multi-core and multiprocessor systems. In: IEEE 13th International Symposium on High Performance Computer Architecture, pp. 13–24 (2007). doi:10.1109/HPCA.2007.346181
10. The Apache Software Foundation: Hadoop (2012). http://hadoop.apache.org. Accessed 1 March 2012
11. Palit, I., Reddy, C.K.: Scalable and parallel boosting with MapReduce. IEEE Trans. Knowl. Data Eng. **24**, 1904–1916 (2012). doi:10.1109/TKDE.2011.208
12. Verma, A., Cho, B.: Breaking the MapReduce stage barrier. In: 2010 IEEE International Conference on Cluster Computing, pp. 235–244 (2010). doi:10.1109/CLUSTER.2010.29
13. Seo, S., Jang, I., Woo, K., Kim, I., Kim, J.-S., Maeng, S.: HPMR: prefetching and pre-shuffling in shared MapReduce computation environment. In: IEEE International Conference on Cluster Computing and Workshops, pp. 1–8 (2009). doi:10.1109/CLUSTR.2009.5289171
14. Zaharia, M., Konwinski, A., Joseph, A.D., Katz, R., Stoica, I.: Improving MapReduce performance in heterogeneous environments. In: Proceedings of the 8th USENIX Conference on Operating Systems Design and Implementation, pp. 29–42 (2008)
15. Zaharia, M., Borthakur, D., Sarma, J.S., Elmeleegy, K., Shenker, S., Stoica, I.: Job scheduling for multi-user MapReduce clusters. (2009). http://www.eecs.berkeley.edu/Pubs/TechRpts/2009/EECS-2009-55.pdf. Accessed 1 March 2012
16. http://hadoop.apache.org/mapreduce/docs/r0.21.0/capacity_scheduler.html (2011). Accessed 1 March 2012
17. Alexandraki, A., Paterakis, M.: Performance evaluation of the deadline credit scheduling algorithm for soft-real-time applications in distributed video-on-demand systems. Clust. Comput. **8**(1), 61–75 (2005). doi:10.1007/s10586-004-4437-4
18. Sandholm, T., Lai, K.: Dynamic proportional share scheduling in hadoop. In: Proceedings of the 15th Workshop on Job Scheduling Strategies for Parallel Processing, pp. 110–131 (2010)
19. Zaharia, M., Borthakur, D., Sarma, J.S., Elmeleegy, K., Shenker, S., Stoica, I.: Delay scheduling: a simple technique for achieving locality and fairness in cluster scheduling. In: Proceedings of the 5th European Conference on Computer Systems, pp. 265–278 (2010)
20. Xie, J., Yin, S., Ruan, X.J., Ding, Z.Y., Tian, Y.: Improving MapReduce performance through data placement in heterogeneous hadoop clusters. In: IEEE International Symposium on Parallel & Distributed Processing, Workshops and PhdForum, pp. 1–9 (2010). doi:10.1109/IPDPSW.2010.5470880
21. Zhang, X.H., Zhong, Z.Y., Feng, S.Z., Tu, B.B., Fan, J.P.: Improving data locality of MapReduce by scheduling in homogeneous computing environments. In: IEEE 9th International Symposium on Parallel and Distributed Processing with Applications, pp. 120–126 (2011). doi:10.1109/ISPA.2011.14
22. Aboulnaga, A., Wang, Z., Zhang, Z.Y.: Packing the most onto your cloud. In: Proceedings of the First International Workshop on Cloud Data Management, pp. 25–28 (2009). doi:10.1145/1651263.1651268
23. Morton, K., Balazinska, M., Grossman, D.: Paratimer: a progress indicator for mapreduce DAGs. In: Proceedings of the 2010 International Conference on Management of Data, pp. 507–518 (2010). doi:10.1145/1807167.1807223
24. Kc, K., Anyanwu, K.: Scheduling hadoop jobs to meet deadlines. In: IEEE Second International Conference on Cloud Computing Technology and Science, pp. 388–392 (2010). doi:10.1109/CloudCom.2010.97
25. Polo, J., Carrera, D., Becerra, Y., Torres, J., Ayguade, E., Steinder, M., Whalley, I.: Performance-driven task co-scheduling for MapReduce environments. In: IEEE Proceedings of Network Operations and Management Symposium, pp. 373–380 (2010). doi:10.1109/NOMS.2010.5488494
26. Phan, L.T.X., Zhang, Z., Loo, B.T., Lee, I.: Real-time MapReduce scheduling. http://repository.upenn.edu/cgi/viewcontent.cgi?article=1988&context=cis_reports (2010). Accessed 1 March 2012
27. Qin, X., Jiang, H., Manzanares, A., Ruan, X., Yin, S.: Dynamic load balancing for IO-intensive applications on clusters. ACM Trans. Storage **5**(3), 1–38 (2009). doi:10.1145/1629075.1629078
28. Herodotou, H.: Hadoop Performance Models (2011). http://www.cs.duke.edu/starfish/files/hadoop-models.pdf. Accessed 1 March 2012
29. Dong, X., Wang, Y., Liao, H.: Scheduling mixed real-time and non-real-time applications in MapReduce Environment. In: IEEE 17th International Conference on Parallel and Distributed Systems, pp. 9–16 (2011). doi:10.1109/ICPADS.2011.115

Zhuo Tang born in 1981. Assistant professor. Received his Ph.D.'s degree in computer science from Huazhong University of Science and Technology, Wuhan, China, in 2008. His current research interests include distributed systems and security of distributed systems.

Junqing Zhou born in 1985. He is currently a Master student in department of computer science and technology at Hunan University, Changsha, China. His research interests include data-intensive computing and distributed systems.

Kenli Li born in 1971. Ph.D. supervisor. Received his Ph.D.'s degree in computer science from Huazhong University of Science and Technology, Wuhan, China, in 2003. He has been a professor of Hunan University since 2007. His current research interests include parallel and distributed computing.

Ruixuan Li born in 1974. He is a professor of Huazhong University of Science and Technology, Wuhan, China. Received his Ph.D.'s degree in computer science from Huazhong University of Science and Technology. His current research interests include distributed system, information search, and security of distributed systems.