Journal of Information Hiding and Multimedia Signal Processing ©2015 ISSN 2073-4212
Ubiquitous International Volume **6**, Number **6**, November **2015**

Shareability and Locality Aware Scheduling Algorithm in Hadoop for Mobile Cloud Computing

Hsin-Wen Wei[1], Tin-Yu Wu[2], Wei-Tsong Lee[1], Che-Wei Hsu[3]

[1]Department of Electrical Engineering
[3]Department of Information Management
Tamkang University
No. 151, Yingzhuan Rd., Tamsui Dist., New Taipei City, 251, Taiwan
{hwwei, wtlee}@mail.tku.edu.tw; kuma0928002898@gmail.com

[2]Department of Computer Science and Information Engineering
National ILan University
No. 1, Sec. 1, Shen-Lung Road, I-Lan, 26047, Taiwan
tyw@niu.edu.tw

Received March, 2015; revised July, 2015

Abstract. *Using different scheduling algorithms can affect the performance of mobile cloud computing using Hadoop MapReduce framework. In Hadoop MapReduce framework, the default scheduling algorithm is First-In-First-Out (FIFO). However, the FIFO scheduler simply schedules task according to its arrival time and does not consider any other factors that may have great impact on system performance. As a result, FIFO cannot achieve good performance in Hadoop for mobile cloud computing. In this paper, we propose a novel scheduling algorithm, called FSLA (FIFO with Shareability and Locality Aware). FSLA is a FIFO-based scheduling policy that considers locality of required data and data sharing probability between tasks. The tasks requesting the same data can be gathered, easily batch processed, and thus reduce the overhead of transferring data between data nodes and computations nodes. The simulation results show that compared to FIFO, FSLA can reach* 65% *improvement in system performance.*

Keywords: Hadoop MapReduce, Shareability, Locality aware scheduling algorithm, Mobile cloud computing.

1. **Introduction.** With the popularization of mobile devices and the increasing demand from end-users, mobile application development evolves quickly. However, mobile devices have limited computing capabilities, battery life and storage space. Therefore, the issues of how to integrate the resources of mobile devices with mobile cloud computing to provide better service have received much attention. At present, mobile cloud computing applications can be generally divided into mobile commerce, mobile learning, mobile health, mobile gaming and other applications [7]. In these applications, mobile cloud plays a very important role especially when mobile users perform web searches by keywords, sounds or tags. A previous research even proposed a mobile monitor system architecture that combines free view point images with a real-time video system to display the status of the residences in a house [16]. To support mobile cloud computing, MapReduce [4] model is often used in these applications for computing. Nevertheless, the default scheduling algorithm in Hadoop MapReduce framework is First-In-First-Out (FIFO), which schedules the tasks according to their arrival time [10]. Under certain circumstances, FIFO is not the best choice [14]. A FIFO scheduler may spend long period of time on one single

task so that another task that can be done quickly may have to wait until the previous one is finished. As a result, the waiting time of some tasks might be too long in the FIFO schedule. Such a scheduling algorithm is not suitable for mobile cloud computing. In addition, using the MapReduce framework, the system usually needs to read external files before performing a task. In the era of big data, the system may spend much more time on I/O than on computing. Therefore, Agrawal et al. studied how best to schedule scans of large data files, in the presence of many simultaneous requests to a common set of files. By sharing scans of the same file as aggressively as possible, the authors tried to maximize the overall rate of processing these files without imposing undue wait time on individual tasks. The simulation results show that their proposed scheduling method can reach shorter execution time [1]. Moreover, the network bandwidth is also an important resource for the MapReduce environment and mobile cloud computing [4]. When a task needs to read a file over a network path, it takes more time than reading the file locally. Consequently, the performance is degraded and the burden on the network is increased. To conserve network bandwidth and improve the system performance, Dean J. et al. proposed to store the input data on the local disks of the machines and schedule a map task on a machine that contains a replica of the corresponding input data [4]. To sum up, in a system coupled with a MapReduce implementation, multiple tasks that require the same file can be executed together for performance enhancements. To schedule a task on a machine that has a copy of the corresponding data can improve system performance as well. However, as far as we know, there is no research that simultaneously considers data shareability among tasks and data locality for performance improvements of a system coupled with a MapReduce implementation. Therefore, to base itself on the FIFO scheduling algorithm and to consider shareability and locality of data, this paper proposes a novel scheduling algorithm, FSLA (FIFO with Shareability and Locality Aware). The simulation results show that our proposed FSLA algorithm not only reduces the time a task spends on reading a file and the network load, but also enhances the system performance. The remainder of this paper is organized as follows. Section 2 introduces the background and related work of this paper. Section 3 describes the proposed FSLA scheduling algorithm. The simulation results are given in Section 4 and the conclusion is given in Section 5.

2. Related Work.

Hadoop is an open-source platform using the MapReduce framework for distributed computing [10]. A Hadoop system includes a MapReduce engine and a Hadoop File System (HDFS). For tasks to find the file locations quickly, HDFS stores all the data in Hadoop. HDFS comprises a single NameNode and a cluster of DataNodes. The NameNode, i.e., the HDFS manager, has to record where the files are stored while the DataNodes stores data blocks in HDFS. HDFS clients can ask the NameNode for a list of DataNodes that own copies of the blocks of a file and access one of the DataNodes directly for the desired blocks. Thanks to the design of HDFS, a Hadoop system can find out the corresponding files quickly while receiving a task request.

To improve MapReduce performance, previous studies have been presented from different perspectives. For distributed computing, a large amount of machines are often utilized. Therefore, some studies aimed at improving energy efficiency and reducing total energy consumption. In [13], the authors propose a framework for systematically considering working node power down strategies with respect to energy consumption and workload response time. Some studies tried to adjust the architecture of MapReduce to handle massive requests since the performance may be degraded as the required data set is blocked between mappers and reducers. In [3], the authors propose a modified architecture to allow required data to be pipelined between operators. Some of them concluded that I/O operations have great impact on system performance [11, 15, 17]. Thus, some researches proposed structured data format and storage [6, 8], and some attempted to enhance the capability of MapReduce-based systems to read the structured input data for improving the I/O performance [5, 15]. In [6], Hadoop++ utilizes indexing technique to improve parallel data access without changing Hadoop framework.

Our study focuses on improving the system efficiency and therefore we will further investigate the scheduling algorithms. Different scheduling systems use different scheduling strategies and a task thus can be assigned a very different priority. Because of different priority arrangements, some tasks with long execution may be resource-wasting and preventing other tasks from being executed. Therefore, using different scheduling algorithms can affect the performance of mobile cloud computing on Hadoop MapReduce framework. To improve the efficiency of system, many scheduling algorithms have been proposed formerly. Some of them try to classify jobs into different categories based on jobs' features and probabilities to make the scheduler be workload-aware and some of them consider the scheduling issue in heterogeneous environment [9, 18, 19]. However, it is hard to classify jobs into appropriate categories. Therefore, features of jobs are not considered in our algorithm and below we give two types of scheduling algorithms that are correlated with our proposed FSLA algorithm.

1) Shared-Scan

Shared-Scan is a scheduling algorithm presented by Agrawal et al. There are many scheduling algorithms in the MapReduce framework. Hadoop allows its users to define the scheduling algorithms themselves for performance improvements. When tasks need to read files and the file size gets bigger and bigger, a system often has to spend much time on reading instead of computing a file. In other situations, multiple pending tasks may require the same file. The core idea of Shared-Scan is to execute multiple tasks that require the same file simultaneously to reduce the waiting time of tasks for performance enhancements [1]. In [1], the authors try to maximizing overall system efficiency while minimizing the maximum waiting time of individual job in the system. They, utilize queuing theory to formulate the waiting time of jobs and provide a hybrid scheduling policy that may balance the overall system efficiency and waiting time of individual job.

2) Location-aware

Location-aware algorithm concerns where the files are stored. When a task is submitted to the system, a location-aware scheduler may assign the task to the machine that holds the replica of the corresponding data to reduce the transmission time and network load. Such a scheduling algorithm can avoid the network congestion due to data transmission and reduce the time a task spends on reading a file [2, 12]. In Hadoop system, data may be replicated and distributed among the nodes, if there is more than one node that can provide required data, how to assign the task to a working node is an interesting issue. To solve the issue, in [2], the system assign weight to data which is required for computing task and calculate the weight of node for data interference according to summarize the weight of data in this node. Then, it will pick up the node with smallest weight and assign the task to the node to reduce data transmission time and to achieve better system performance.

However, the above-mentioned two scheduling algorithms do not consider the order of task arrivals, which may lead to indefinite task waiting times. Moreover, in mobile cloud computing environment, we need to consider the view point of users. Users always expect to be treated fairly and first-come-first-serve policy is usually identified to be a fair algorithm from users' perspective. For these reasons, we integrate FIFO with the benefits of the above two scheduling algorithms and propose FSLA. FSLA is a method that not only considers data shareability and locality aware, but also allocates computing resources according to task arrivals, without imposing lengthy waiting time on tasks.

3. FIFO WITH SHAREABILITY AND LOCALITY AWARE ALGORITHM.

The goal of proposed FSLA is to consider data shareability, data locality, and the arrival time of task simultaneously to reducing the data transferring time and waiting time. Therefore, to achieve the goal of FSLA, there are two kinds of queues in the system: base queue and waiting queue, which store tasks according to their arrival times as shown in Figure 1. The base queue stores the pending tasks in the system. When a machine is idle, a task in the base queue can be assigned to the machine and executed. The waiting queue stores the tasks that are waiting for others requesting the same data. To reduce transmission time and utilize the benefits of caching, these tasks are executed on a single machine and called a task set.

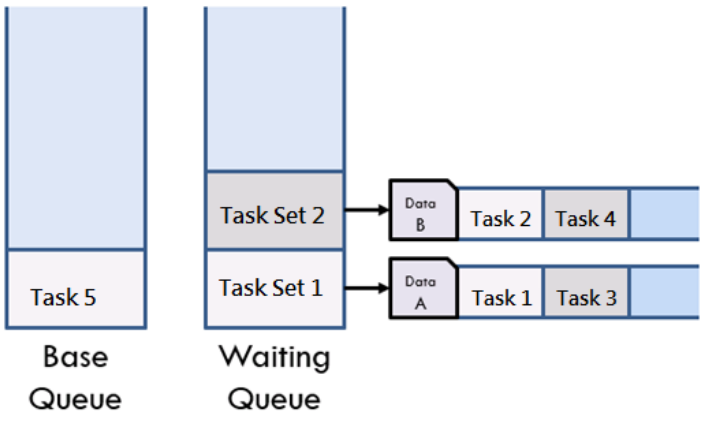


FIGURE 1. Illustration of base queue and waiting queue.

When a new task is submitted to the system, our proposed FSLA determines whether to put the task into base queue or waiting queue. In other words, the task is executed in FIFO way or the task needs to wait until the arrival of other tasks. Before making the decision, the system has to obtain the following information of the task first: (1) *estimated task file transmission time*, which is the time spending on transmitting file that is requested by the task and (2) *estimated number of tasks in the task set*, which is the number of tasks may request the same file within a time interval. Based on the two parameters, the system reckons whether the task should wait for other tasks. If yes, the system defines a waiting time limit for the task and puts the task into waiting queue; otherwise, the system puts the task into base queue. Further details will be given in the following subsections.

3.1. Task Information.

To determine which queue a newly arrival task should be put into and compute the waiting time limit for the task that needs to wait for other tasks, one has to know the task file transmission time first. We list the symbols and their descriptions that will be used in the following equations in Table 1. In the MapReduce framework, before the system performs the map function, pending tasks have to inform the system their desired data for computation resource allocation. In Hadoop, after receiving the request from a task, the system uses HDFS to find out the locations of the corresponding files. In this paper, as shown in Table 1, t denotes the transmission time of a file. $t = D/e$, where D is the file size and e is the file I/O speed. After obtain the estimated file transmission time, the system has to estimate the average arrival rate of tasks that may request the same file, which is called estimated task arrival rate, denoted by Δ, and computed in Equation 1.

$$\Delta = \lambda \times p \tag{1}$$

λ means the average task arrival rate in the system. p denotes the probability that tasks request the same file and we call it data sharing probability. λ multiplied by p gives the estimated task arrival rate.

Table 1. Symbols of Task Information

Symbol	Description
n	Number of tasks in the task set
C_i	Computation time of the task i
λ	Task arrival rate
p	Data sharing probability
Δ	Estimated task arrival rate
D	File Size
e	File I/O Speed
t	File transmission time
ω	Maximum Waiting Time
ω_p	Predefined Waiting Time
ω_s	Suggested Waiting Time
T_1	Average completion time of n tasks (without waiting)
T_2	Average completion time of n tasks (with waiting)

Next, we can use Equation 2 to obtain the estimated number of tasks in the task set.

$$n = \lfloor 1 + \Delta \times \omega_s \rfloor \tag{2}$$

To multiply Δ, the estimated task arrival rate, by $omega_s$, the suggested waiting time, plus 1, and select an integral number smaller than the calculation result, we can get the estimated number of tasks in the task set. When the suggested waiting time equals 0 or $\Delta \times \omega_s \leq 1$, at least a default task needs to be executed in the task set. This is the reason of plus 1 in the equation.

With the file transmission time and the estimated number of tasks in the task set, we can perform further calculation and decide whether the task should wait for other tasks or not. Assume that C_i denotes the computation time of a task i, n denotes the number of tasks that require the same file, and the file transmission time is t. If not waiting, for each task i, the task completion time equals to $t + C_i$, i.e., file transmission time plus computation time. Note, other overheads in handling a task in the system is ignored here. Supposing there are n tasks in the systems, the average completion time of n tasks, T_1, can be given by Equation 3.

$$T_1 = t + \frac{\sum_{i=1}^{n} C_i}{n} \tag{3}$$

If the task is suggested to wait for $omega_s$, all tasks in the task set share the same file transmission time t. Therefore, the average completion of n tasks, T_2, can be given by Equation 4.

$$T_2 = \omega_s + \frac{t + \sum_{i=1}^{n} C_i}{n} \tag{4}$$

For the final task in the task set, it may arrive simultaneously with the first task in a worst-case scenario. Therefore, the final task has to wait for ω_s seconds like the first task before being executed. Here, we consider the worst situation: all tasks in the task set arrive at the same time. Next, the scheduler compares the values of T_1 and T_2. If $T_1 \le T_2$, it means that to wait for other tasks results in the longer average completion time. Then, the task should not wait for other tasks but be executed directly. If $T_1 \ge T_2$, it means the task should wait. Based on such a condition, our proposed FSLA can decide whether a task should wait and the equation can be simplified as Formula 5. In other words, if the file transmission time of a task is longer than the suggested waiting time for other tasks requiring the same file, the task should be put into the waiting queue.

$$T_1 \ge T_2 \Rightarrow t \ge \omega_s + \frac{t}{n} \Rightarrow t \ge \frac{1}{\Delta} \tag{5}$$

In our algorithm, we can use $T_1 - T_2$ to know the time difference between waiting and no waiting situations. The larger the time difference is, the better the efficiency improvement will be. To compute the maximum value of $T_1 - T_2$, we can get the optimum of ω_s. Let $f(\omega_s) = T_1 - T_2$. Solve the differential equation $f'(\omega_s) = 0$, the optimum of ω_s can be given by Equation 6 and 7.

$$\omega_s = t - \omega_s - \frac{t}{1 + \lambda \cdot p \cdot \omega_s} \tag{6}$$

$$\Rightarrow \omega_s = \frac{\sqrt{\lambda \cdot p \cdot t} - 1}{\lambda \cdot p} \tag{7}$$

Since the users may want to limit the waiting time of a task, our proposed algorithm allows users to set the predefined waiting time ω_p for service quality guarantee. The predefined waiting time ω_p is a fixed constant. We select a minimum value between ω_s and ω_p as the maximum waiting time limit. Therefore, the maximum waiting time ω can be computed by Equation 8.

$$\omega = \min(\omega_s, \omega_p) \tag{8}$$

With the above information, the proposed FSLA can schedule tasks properly. Normally, to schedule tasks, scheduling decision needs to be made in the following two situations: when a new task is submitted into the system and when there is an idle machine in the system. Next, we will discuss the two different situations, respectively.

3.2. **Scheduling Mechanism.**

3.2.1. *When a new task arrives.* When a new task arrives in the system, the scheduler first checks whether the base queue and the waiting queue are empty. If there is a task in a queue, the new task is put into the base queue or put into the corresponding existing waiting queue based on the waiting time calculated via Formula 5 and Equation 7, and if there is no corresponding waiting queue; the scheduler creates a new waiting queue for the task. Otherwise, both queues are empty, then the scheduler checks if there is an available machine to execute the task. If there is an idle machine that has the corresponding file the task needs, the scheduler assign the machine to executes the task directly. If there is an idle machine but does not have the corresponding file on it, the scheduler starts to compute whether the task should wait for other tasks. If not, the scheduler assign the machine to executes the task directly; otherwise, create a new waiting queue for the task. Figure 2 displays the detailed flowchart and Algorithm 1 describes how the scheduler handles a new arrival task. Table 2 shows the symbols used in Algorithm 1.

Table 2. Symbols used in algorithms

Symbol	Description
τ_i	Task i
F_i	The file requested by task τ_i
D_i	The size of F_i
S_i	The task set, in which all tasks request the same file with τ_i
B	Base queue
W	Waiting queue

Algorithm 1 A new task τ_i arrives in the system

```
 1: Begin
 2: if $B$ and $W$ are empty then
 3:   if there is an idle machine then
 4:     if $F_i$ is located in the idle machine then
 5:       assign $\tau_i$ to the idle machine
 6:     else /* $F_i$ is not located in the idle machine */
 7:       if $\frac{D_i}{e} > \frac{1}{\Delta}$ then
 8:         compute waiting time for $\tau_i$;
 9:         create new $S_i$;
10:         put $\tau_i$ into $S_i$;
11:         add $S_i$ into $W$;
12:       else
13:         assign $\tau_i$ to the idle machine;
14:       end if
15:     end if
16:     return;
17:   end if
18: end if
          /* base queue or waiting queue is not empty; or no idle machine */
19: if $\frac{D_i}{e} > \frac{1}{\Delta}$ then
20:   compute waiting time for $\tau_i$
21:   if $S_i$ exists in $W$ then
22:     put $\tau_i$ into $S_i$
23:   else
24:     create new $S_i$;
25:     put $\tau_i$ into $S_i$;
26:     add $S_i$ into $W$;
27:   end if
28: else /* $\tau_i$ should not wait for other tasks */
29:   push $\tau_i$ into $B$;
30: end if
31: End
```

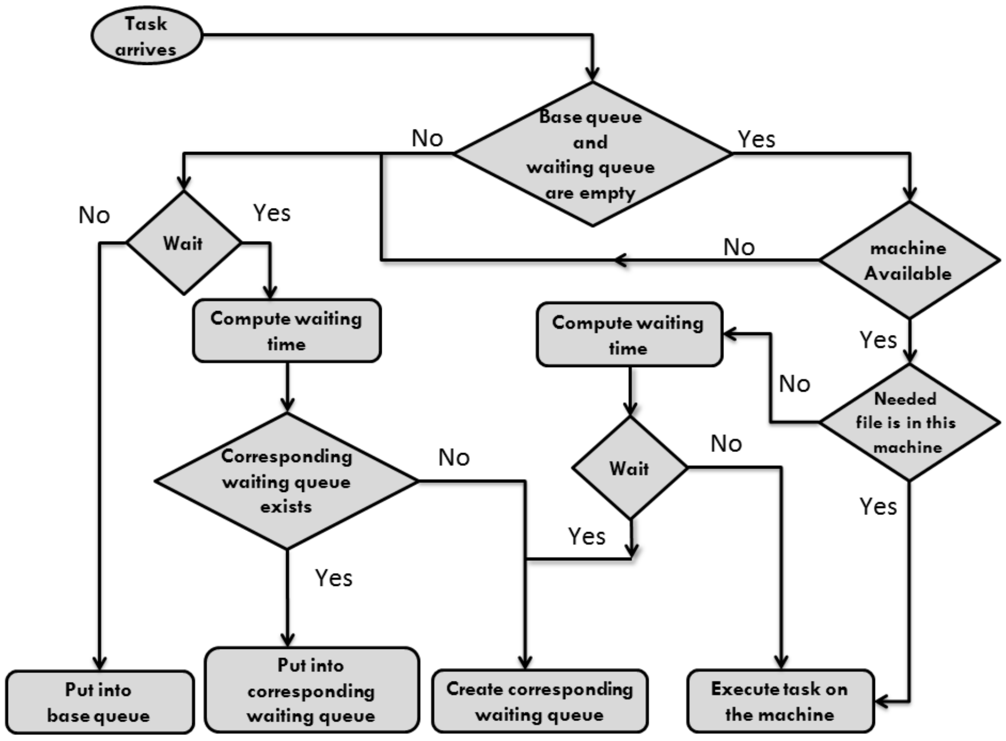

Task arrives
Base queue and waiting queue are empty
No
Yes
No
Wait
Yes
No
machine Available
Compute waiting time
Compute waiting time
Yes
No
Needed file is in this machine
Corresponding waiting queue exists
No
Wait
No
Yes
Yes
Yes
Put into base queue
Put into corresponding waiting queue
Create corresponding waiting queue
Execute task on the machine

Figure 2. Illustration of the scheduler in handling a new arrival task

The task in the waiting queue has to wait for other tasks requiring the same file to be put into the same task set and executed altogether. When the scheduler decides whether a task should wait, special considerations must be given to file transmission time, estimated number of tasks in the task set, waiting time and whether waiting is worthwhile.

3.2.2. *When there is an idle machine in the system.* When there is an idle machine in the system, the scheduler first finds the task with earliest arrival time in the base queue and the waiting queues. If the task belongs to the base queue, the scheduler assigns the task to the machine and the machine executes the task directly. If the task belongs to the waiting queues, the scheduler checks if this idle machine has the file the task needs or the selected task reaches the waiting time limit. If yes, similarly assigns the task to the machine and execute it. Otherwise, the scheduler finds the next task to execute from the base queue and the waiting queues. Figure 3 and Algorithm 2 show the detailed flowchart and algorithm that handle this situation. Next, we will explain how the scheduler in our proposed FSLA scheduling algorithm decides whether a task should wait for other tasks. Assume that the speed e of reading a file in a machine is 1GB per second and the file size D of file F is 9GB. For a task τ_i that needs the file F, the file transmission time t will be 9 second. Assume the task arrival rate λ in the system is 2, which means there are 2 tasks entering the system per second, and the probability that the tasks require the same file F is 50%, the estimated task arrival rate Δ will be 1.

When a task τ_i arrives in the system, the scheduler reckons whether task τ_i should wait for other tasks. As mentioned previously, the system uses Formula 5 to decide whether a task should wait. In this example, the condition can take place and the scheduler puts the task τ_i into the waiting queue. At this time, the scheduler has to compute the maximum waiting time. Using Equation 7, we can obtain that the optimal waiting time ω_s for task τ_i is 2.

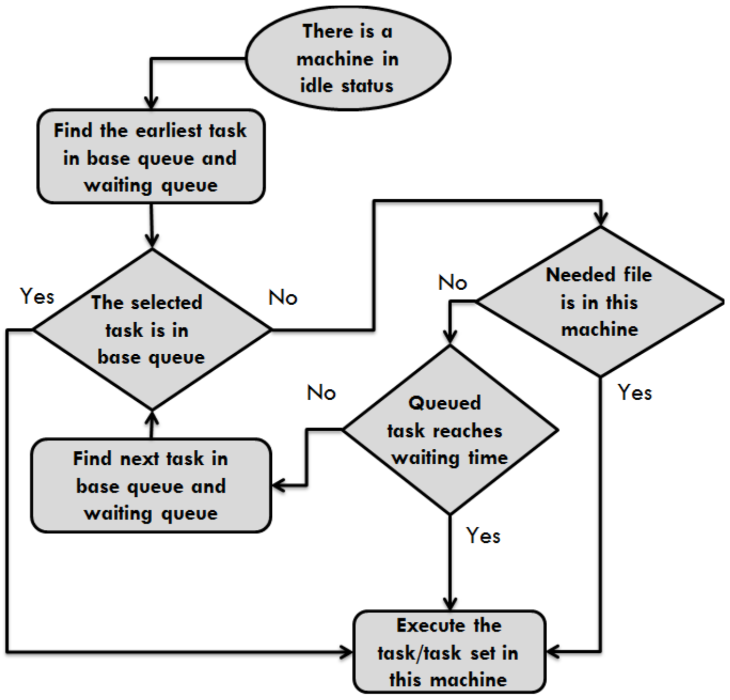

There is a machine in idle status
Find the earliest task in base queue and waiting queue
The selected task is in base queue
Yes
No
Needed file is in this machine
No
Yes
Queued task reaches waiting time
No
Yes
Find next task in base queue and waiting queue
Execute the task/task set in this machine

FIGURE 3. When there is an idle machine in the system

Algorithm 2 A machine is idle

```
 1: Begin
 2: let $B' = B$, $W' = W$;
 3: if $B'$ or $W'$ is not empty then
 4:   Find the earliest arrived task among all tasks in $B'$ and $W'$, and denote the task as $\tau_e$;
 5: end if
 6: while $B'$ or $W'$ is not empty and there is a machine $m$ in idle do
 7:   if $\tau_e \in B'$ then  /* $\tau_e$ is in the base queue */
 8:     assign $\tau_e$ to $m$;
 9:     $B' = B' - \{\tau_e\}$;  /* remove the task $\tau_e$ from base queue */
10:   else  /* $\tau_e$ is in the waiting queue */
11:     if $F_i$ is located in $m$ then
12:       assign all tasks in $S_i$ to $m$;
13:       $W' = W' - \{S_i\}$;  /* remove all tasks in $S_i$ from waiting queue */
14:     else  /* $F_i$ is not located in $m$ */
15:       if the time of tasks queued in $S_i$ reach their maximum waiting time then
16:         assign all tasks in $S_i$ to $m$;
17:         $W' = W' - \{S_i\}$;
18:       else
        /* select next task but does not remove the previous task from waiting queue */
19:         let $W'' = W' - \{\tau_e\}$;
20:         let $\tau_e$ be the next earliest arrived task in $B'$ and $W''$;
21:       end if
22:     end if
23:   end if
24: end while
25: End
```

Supposing the predefined waiting time ω_p set by user is 3, then, the waiting time limit ω can be easily obtained by Equation 8 as follows:

$$\omega = \min(2, 3) = 2$$

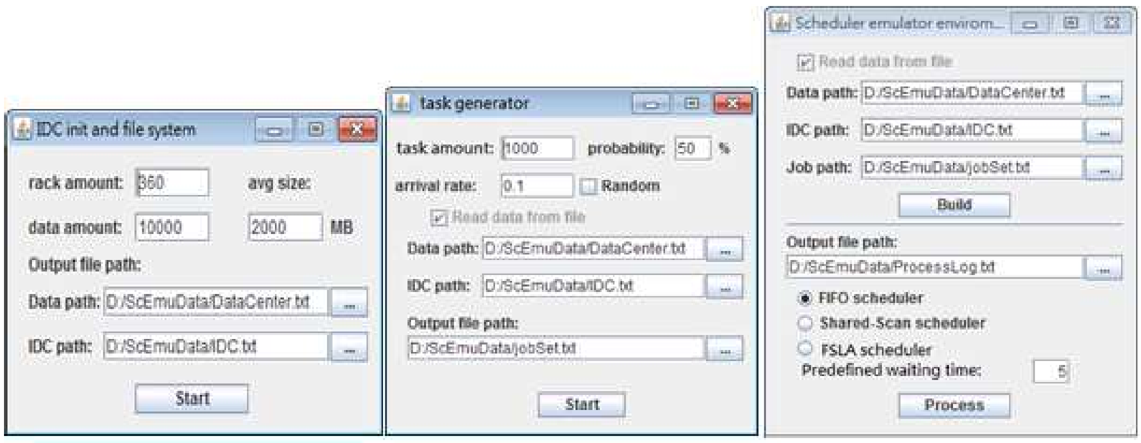
With the waiting time limit, the estimated number of tasks in the task set can be computed by Equation 2. In this example, the estimated number of tasks in the task set is 3. Therefore, when the system needs to execute 3 tasks, the average completion time of all tasks without waiting is $9 + \epsilon$ (i.e., T_1) and the average completion time of all tasks while waiting is $5 + \epsilon$ (i.e., T_2). Please notice that ϵ is the average task computation time. In such a scenario, our proposed FSLA will put task τ_i into the waiting queue and wait for other tasks requiring the same file since the calculations above reveal that to wait for other tasks requiring the same file is more beneficial than to execute the task directly.

4. **Simulation Results.**

In this section, the scheduling algorithms, including FIFO, Shared-Scan and our proposed FSLA, will be compared in the same environment using the same data set to compute the average absolute perceived waiting time (AAPWT) for performance comparison. The average absolute perceived waiting time (AAPWT) of a task means the time that a task queued in the system, i.e., the time spending on waiting for execution. Agrawal et al. uses the average absolute perceived waiting time as the performance metric in their study [1]. For consistency, this paper also uses the AAPWT as the performance metric.

4.1. **Simulation Environment.**

In this paper, the scheduling simulation system is developed with JAVA programming language and consists of three main parts: 1) computing center module, 2) task generator module and 3) the scheduler module as shown in Figure 4. Computing center module is responsible for generating datasets and setting the computing resources for handling the requested tasks. To generate different datasets and configure computing resources settings, three variables are used: number of racks, data amount, and average data size. The number of racks determines how many machines and storages are in our simulated computing center. The data amount and the average data size determines the numbers of files and the file size. Assume that there are 8 compute units in each rack, each compute unit has a storage capacity of 3TB and each rack has additional storage capacity of 288TB shared among 8 compute units. Therefore, each rack has a storage capacity of 312TB in total. In our simulation environment, there are 180 racks, i.e. 1440 compute units and a capacity of 800PB. The read/write speed of each compute unit is set to 1024Mb/sec. and the network read speed is set to 500Mb/sec. In our simulation, if the file the task needs is in the storage space of the compute unit or in the rack where the compute unit belongs to, the scheduler executes the task on this machine using the read/write speed of the compute unit. If not, the task is regarded to be executed on the remote machine using the network read speed. Task generator module is responsible for generating task sets for the scheduler to test. To generate different task sets, the scheduler allows users to define four variables: 1) *task amount*, based on which the module generates enough test tasks; 2) *arrival rate*, which controls the arrival time of tasks based on Poisson distribution; 3) *average data size*, which defines the averaged size of data required by jobs; 4) *average data sharing probability*, based on which the module generates the corresponding task sets. Note, the data sharing probability of each task is uniformly distributed. The scheduler module is responsible for running the scheduling simulation using the datasets generated by the above mentioned computing center module and task generator module. Also, the scheduler records the simulation results for the subsequent analysis. Detailed descriptions and analysis of the simulation results are given below.


IDC init and file system

rack amount: 360 avg size:

data amount: 10000 2000 MB

Output file path:

Data path: D:/ScEmuData/DataCenter.txt

IDC path: D:/ScEmuData/IDC.txt

Start

task generator

task amount: 1000 probability: 50 %

arrival rate: 0.1 ☐ Random

☑ Read data from file

Data path: D:/ScEmuData/DataCenter.txt

IDC path: D:/ScEmuData/IDC.txt

Output file path:

D:/ScEmuData/jobSet.txt

Start

Scheduler emulator envirom...

☑ Read data from file

Data path: D:/ScEmuData/DataCenter.txt

IDC path: D:/ScEmuData/IDC.txt

Job path: D:/ScEmuData/jobSet.txt

Build

Output file path:

D:/ScEmuData/ProcessLog.txt

● FIFO scheduler

○ Shared-Scan scheduler

○ FSLA scheduler

Predefined waiting time: 5

Process

FIGURE 4. GUI of three modules of the simulation system

4.2. **Simulation Results.** In this paper, we adopt four control variables as the criterion to generate test datasets: task amount, average arrival rate, average data size and average data sharing probability. We conduct the scheduling simulation using different test datasets and analyze the performance improvement of FSLA as follows.

1) Task amount

Figure 5 shows the scheduling simulation on task amount and reveals that the increase of task amount obviously has an impact on FIFO, but not on Shared-Scan and our proposed FSLA because Shared-Scan and FSLA both have batch process design and thus are unaffected when the task amount gets larger. Our proposed FSLA reaches 27%-87% improvement compared with FIFO and 1.73% – 2.27% improvement compared with Shared-Scan.

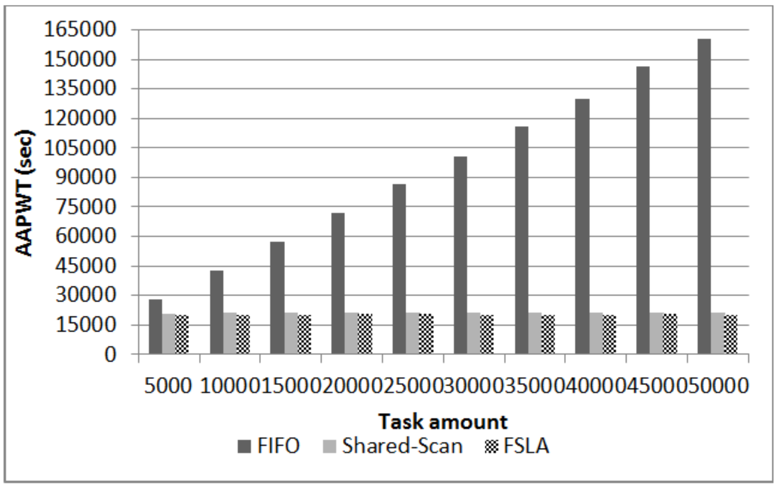


FIGURE 5. Simulation result of different task amounts

2) Arrival Rate

Figure 6 shows the scheduling simulation on task arrival rate and reveals that our proposed FSLA has a lower AAPWT than FIFO and Shared-Scan. FSLA reaches 3% – 86% improvement compared with FIFO and 0.75% to 5.7% improvement compared with Shared-Scan. We found that when the arrival rate is low, the performance improvement of FSLA is quite limited. But, when the arrival rate gets higher, the performance improvement becomes better.

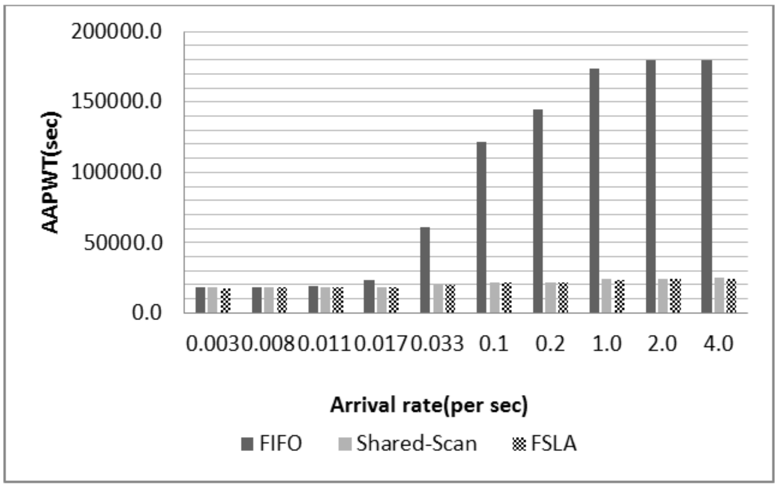


Figure 6. Simulation result under various arrival rates

3) Average data size

Figure 7 shows the scheduling simulation on the average data size. The average data size starts from 4GB and doubles until it reaches 1TB. The figure reveals that when the average data size becomes bigger, the AAPWT of FIFO increases very quickly but that of FSLA increases slowly. Our proposed FSLA reaches 10% – 79% improvement compared with FIFO and 1.37% – 6.92% improvement compared with Shared-Scan.

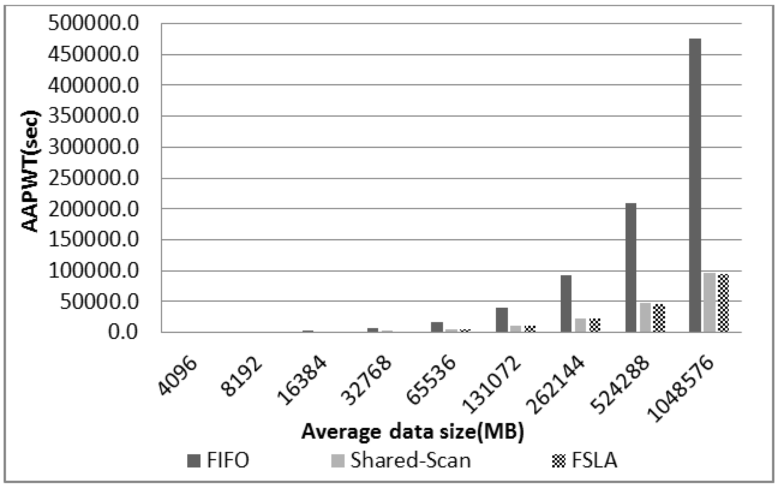


Figure 7. Simulation result under various data sizes

4) Average data sharing probability

Figure 8 shows the scheduling simulation on the average data sharing probability. Compared with FIFO, FSLA reaches 67% – 79% improvement. Compared with Shared-Scan, FSLA reaches 0.83% – 2.78% improvement. When the average data sharing probability $> 0.1\%$, FSLA obviously outperforms FIFO.

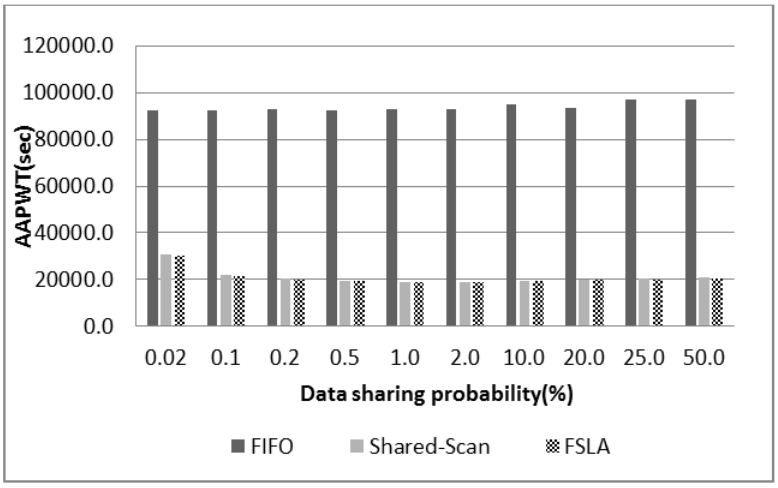


Figure 8. Simulation result of different data sharing probabilities

Previous simulation results show that our proposed FSLA has a lower AAPWT than FIFO and Shared-Scan. Moreover, using no matter the average data sharing probability or the exact data sharing probability to compute the optimum of waiting time has little influence on the performance.

5) Locality-ratio

The locality ratio represents the percentage of tasks among all tasks that are executed in a machine that contains their needed files and without transferring files from network. The locality ratio of algorithm is important because higher locality ratio implies lower network overhead and can benefit from caching mechanism. Here, we compare the locality-ratio of FSLA, FIFO and Shared-Scan in different situations. Figure 9 shows the locality ratio in different task amount and reveals that FSLA has a higher locality ratio than FIFO and Shared-Scan. The change of task amount has little influence on the locality ratio.

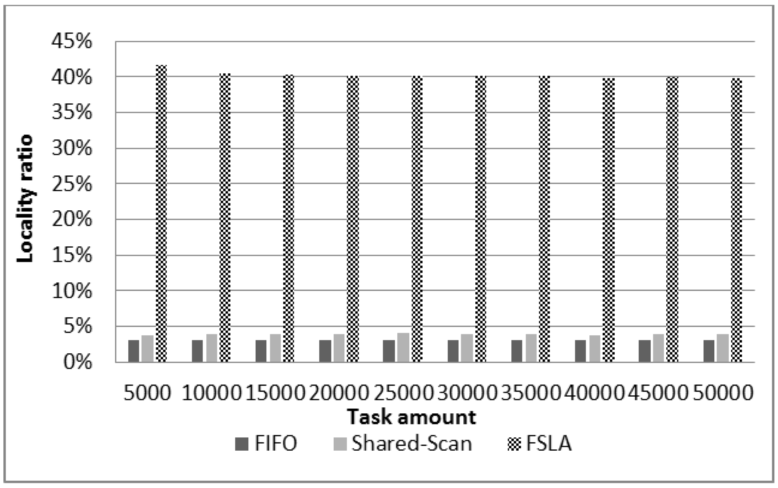


Figure 9. Locality ratio vs. task amount

Figure 10 shows the locality ratio in different arrival rate. The figure reveals that the locality ratio of FSLA gradually decreases when the arrival rate increases because the possibility of idle machines in the system is small. In this way, the probability to execute a task in the machine that contains a replica of the corresponding data is comparatively low. Therefore, the locality ratio decreases when the arrival rate increases. Nevertheless, compared with the other two algorithms, FSLA has an obviously higher locality ratio and thus can decrease the network load substantially.

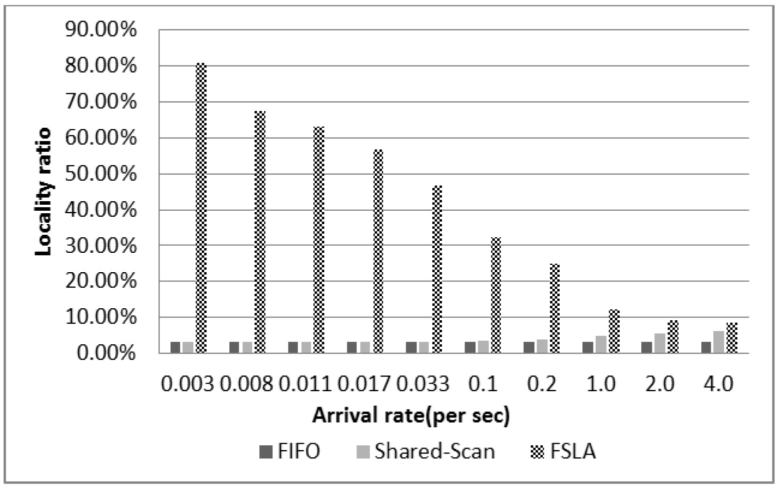


Figure 10. Locality ratio vs. arrival rate

Figure 11 shows the locality ratio under various data size. Similarly, locality ratio of FSLA gradually decreases when the data size increases because the task needs larger data may occupied a machine longer that force other tasks to be executed in other machines without locality concern.

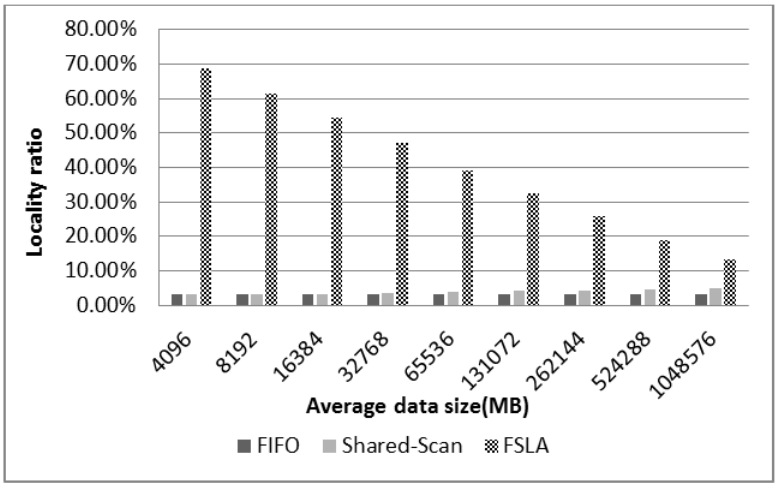


Figure 11. Locality ratio vs. average data size

Figure 12 shows the locality ratio in different data sharing probability. The locality ratio of FSLA declines when data sharing probability is increased. This is because of that while data sharing probability is increased but still not large enough, the possibility of FSLA to put a task into waiting queue is also increased but lead to FSLA to select another task without locality consideration. However, compared with the other two algorithms, FSLA still has an obviously higher locality ratio.

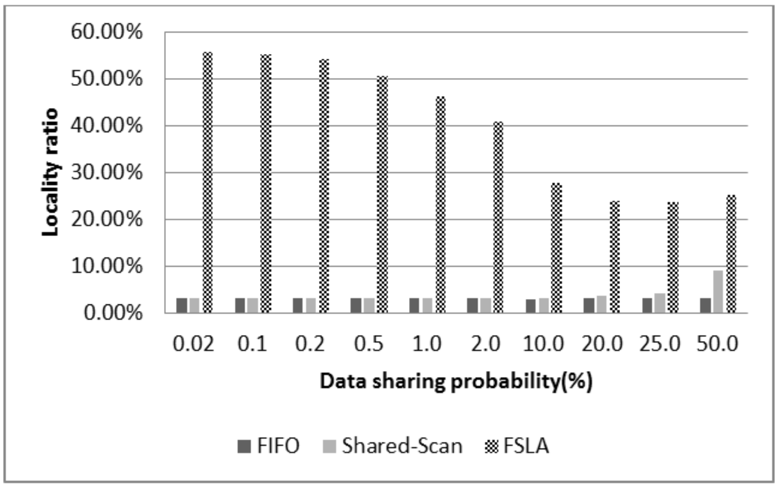


Figure 12. Locality ratio vs. data sharing probability

5. **Conclusion.** In Hadoop framework, different scheduling algorithms have different impact on system performance. Therefore, this paper proposed FSLA, a FIFO-based scheduling algorithm that consider data shareability and data locality to improve the system performance and enhance the benefits of caching mechanism. In FSLA, there are two queues base queue and waiting queue to classify the tasks into two types: one does not need to wait for other tasks and another that needs to wait for other tasks. Through the classification, if the arrival rate of tasks is high and the requested files are large, FSLA outperforms the commonly used FIFO. Assuming the tasks entering the system all require small files, FSLA still performs better than FIFO. According to the AAPWT computed from the algorithms, the simulation results show that AAPWT of the proposed FSLA compared to that of FIFO and Shared-scan is improved in 26.75% − 82.75% and 1.17% − 4.42%, respectively. Moreover, the locality ratio of FSLA is around 40% better than that of FIFO and Shared-Scan. In conclusion, FSLA outperform than others because the proposed FSLA is based on FIFO and further considers data shareability and data locality aware. Using our proposed FSLA in Hadoop framework not only brings good computing efficiency but also reduces the network load.

In this paper, the capability of each node is not considered in the scheduling algorithm and all nodes are assumed to have the same computing power and storage capacity, which might not be true in real world system. Therefore, our future work is to take node's capability into consideration when study job scheduling. Moreover, we will also study the features of applications for improving the system performance and further expand the results to QoS consideration.

Acknowledgment. This work is partially supported by the MOST(Ministry of Science and Technology) research grants MOST-103-2221-E-032-035, MOST-103-2221-E-197-019, MOST-104-2221-E-032-008. The authors also gratefully acknowledge the helpful comments and suggestions of the reviewers, which have improved the presentation.

REFERENCES

[1] P. Agrawal, D. Kifer, & C. Olston, Scheduling shared scans of large data files, *Proc. of the VLDB Endowment*, pp. 958-969, 2008.
[2] T.Y. Chen, H.W. Wei, M.F. Wei, Y.J. Chen, T.s. Hsu, & W.K. Shih, LaSA: A locality-aware scheduling algorithm for hadoop-MapReduce resource assignment, *Proc. of International Conference on the Collaboration Technologies and Systems (CTS)*, pp. 342-346, 2013.

[3] T. Condie, N. Conway, P. Alvaro, J.M. Hellerstein, K. Elmeleegy, & R. Sears, MapReduce online, *Proc. of the 7th USENIX conference on Networked systems design and implementation*, vol. 10, no.4, 2010.

[4] J. Dean, & S. Ghemawat, MapReduce: Simplified data processing on large clusters, *Communications of the ACM*, vol. 51, no. 1, pp. 107-113, 2008.

[5] J. Dean, & S. Ghemawat, MapReduce: A flexible data processing tool, *Communications of the ACM*, vol. 53, no. 1, 72-77, 2010.

[6] J. Dittrich, J. Quiané-Ruiz, A. Jindal , Y. Kargin, V. Setty, & J. Schad, Hadoop++ : Making a yellow elephant run like a cheetah (without it even noticing), *Proc. of the VLDB Endowment*, pp. 515-529, 2010.

[7] H.T. Dinh, C. Lee, D. Niyato, & P. Wang, A survey of mobile cloud computing: architecture, applications, and approaches, *Wireless Communications and Mobile Computing*, vol. 13, no. 18, pp. 1587-1611, 2013.

[8] A. Floratou, J.M. Patel, E.J. Shekita, & S. Tata, Column-oriented storage techniques for MapReduce, *Proc. of the VLDB Endowment*, pp. 419-429, 2011.

[9] S. Gupta, C. Fritz, B. Price, R. Hoover, & J. de Kleer, ThroughputScheduler: Learning to Schedule on Heterogeneous Hadoop Clusters, *Proc. of the International Conference on Autonomic Computing (ICAC '13), 2013.*

[10] Hadoop. Welcome to apache^{TM} hadoop!. http://hadoop.apache.org/

[11] D. Jiang, B.C. Ooi, L. Shi, & S. Wu, The performance of MapReduce: An in-depth study, *Proc. of the VLDB Endowment*, pp. 472-483, 2010.

[12] M.A. Kozuch, M.P. Ryan, R. Gass, S.W. Schlosser, D. O'Hallaron, J. Cipar, & G.R. Ganger, Tashi: Location-aware cluster management, *In the Proc. of the 1st Workshop on Automated Control for Datacenters and Clouds*, pp. 43-48, 2009.

[13] W. Lang, & J. M. Patel, Energy management for mapreduce clusters, *Proc. of the VLDB Endowment*, pp. 129-139, 2010.

[14] K. Lee, Y. Lee, H. Choi, Y.D. Chung, & B. Moon, Parallel data processing with MapReduce: A survey, *ACM SIGMOD Record*, vol. 40, no. 4, pp. 11-20, 2011.

[15] B. Li, E. Mazur, Y. Diao, A. McGregor, & P. Shenoy, A platform for scalable one-pass analytics using MapReduce, *In the Proc. of the 2011 ACM SIGMOD International Conference on Management of Data*, pp. 985-996, 2011.

[16] Y.C. Li, I.J. Liao, H.P. Cheng, & W.T. Lee, A cloud computing framework of free view point real-time monitor system working on mobile devices, *In International Symposium on Intelligent Signal Processing and Communication Systems*, pp. 1-4, 2010.

[17] A. Pavlo, E. Paulson, A. Rasin, D.J. Abadi, D.J. DeWitt, S. Madden, & M. Stonebraker, A comparison of approaches to large-scale data analysis, *In the Proc. of the 2009 ACM SIGMOD International Conference on Management of Data*, pp. 165-178, 2009.

[18] W.z. Sun & X.j. Wang, The Optimization of Hadoop Scheduling Algorithms on Distributed System for Processing Traffic Information, *Proc. of International Conference on Soft Computing Techniques and Engineering Application*, 2013.

[19] Z. Wang, Z.d. Zhu, P.f. Zheng, Q. Liu, & X. Dong, A new scheduler strategy for heterogeneous workload-aware in hadoop, *Proc. of 8th ChinaGrid Annual Conference (ChinaGrid)*, 2013.