Novel Scheduling Algorithms for Efficient Deployment of MapReduce Applications in Heterogeneous Computing Environments

Sun-Yuan Hsieh, *Senior Member, IEEE*, Chi-Ting Chen, Chi-Hao Chen, Tzu-Hsiang Yen, Hung-Chang Hsiao, and Rajkumar Buyya, *Fellow, IEEE*

Abstract—Cloud computing has become increasingly popular model for delivering applications hosted in large data centers as subscription oriented services. Hadoop is a popular system supporting the MapReduce function, which plays a crucial role in cloud computing. The resources required for executing jobs in a large data center vary according to the job type. In Hadoop, jobs are scheduled by default on a first-come-first-served basis, which may unbalance resource utilization. This paper proposes a job scheduler called the *job allocation scheduler* (JAS), designed to balance resource utilization. For various job workloads, the JAS categorizes jobs and then assigns tasks to a *CPU-bound queue* or an *I/O-bound queue*. However, the JAS exhibited a locality problem, which was addressed by developing a modified JAS called the *job allocation scheduler with locality* (JASL). The JASL improved the use of nodes and the performance of Hadoop in heterogeneous computing environments. Finally, two parameters were added to the JASL to detect inaccurate slot settings and create a dynamic job allocation scheduler with locality (DJASL). The DJASL exhibited superior performance than did the JAS, and data locality similar to that of the JASL.

Index Terms—Hadoop, heterogeneous environments, heterogeneous workloads, MapReduce, scheduling

1 Introduction

The scale and maturity of the Internet has recently increased dramatically, providing excellent opportunities for enterprises to conduct business at a global level with minimum investment. The Internet enables enterprises to rapidly collect considerable amounts of business data. Enterprises must be able to process data promptly. Similar requirements can be observed in scientific and Big Data applications. Therefore, promptly processing large data volumes in parallel has become increasingly imperative. Cloud computing has emerged as a new paradigm that supports enterprises with low-cost computing infrastructure on a pay-as-you-go basis.

In cloud computing, the MapReduce framework designed for parallelizing large data sets and splitting them into thousands of processing nodes in a cluster is a crucial concept. Hadoop [1], which implements the MapReduce programming framework, is an open-source distributed system used by numerous enterprises, including Yahoo and Facebook, for processing large data sets.

Hadoop is a server-client architecture system that uses the master-and-slave concept. The master node, called *JobTracker*, manages multiple slave nodes, called *TaskTracker*s, to process tasks assigned by the *JobTracker*. A client submits MapReduce job requests to the *JobTracker*, which subsequently splits jobs into multiple tasks, including map tasks and reduce tasks. Map tasks receive input data and output intermediate data to a local node, whereas reduce tasks receive intermediate data from several *TaskTracker*s and output the final result.

By default, Hadoop adopts a first-come-first-served (FCFS) job scheduling policy: A *TaskTracker* transfers requests to a *JobTracker* through the *Heartbeat cycle*. When the *JobTracker* receives a *Heartbeat* message, it obtains the number of free slots in the *TaskTracker* and then delivers the task to the *TaskTracker*, which executes the task immediately. Because resource utilization is not considered in the default job scheduling policy of Hadoop (FCFS), some slave nodes do not have the capacity to perform an assigned task. Therefore, such nodes cannot continue to execute tasks after the system releases resources, leading to poor performance in a Hadoop system. The resource allocation problem is an NP-complete problem [25]; this type of problem has received substantial attention in cloud computing. In Facebook fair scheduler [4] and Yahoo capacity scheduler [3], multiple queues are used to achieve resource allocation through a policy that entails assigning a distinct number of resources to each queue, thus providing users with various queues to maximize resource utilization. However,

• *S.-Y. Hsieh is with the Department of Computer Science and Information Engineering, Institute of Medical Informatics, Institute of Manufacturing Information and Systems, National Cheng Kung University, No. 1, University Road, Tainan 701, Taiwan. E-mail: hsiehsy@mail.ncku.edu.tw.*
• *C.-T. Chen, C.-H. Chen, T.-Z. Yen, and H.-C. Hsiao are with the Department of Computer Science and Information Engineering, National Cheng Kung University, No. 1, University Road, Tainan 701, Taiwan. E-mail: babababa1108@gmail.com, {kitfretas, robinpkpk0523}@hotmail.com, hchsiao@csie.ncku.edu.tw.*
• *R. Buyya is with the Department of Computing and Information Systems, Cloud Computing and Distributed Systems Laboratory, The University of Melbourne, Australia. E-mail: rbuyya@unimelb.edu.au.*

Manuscript received 3 July 2015; revised 28 Feb. 2016; accepted 15 Mar. 2016. Date of publication 11 Apr. 2016; date of current version 5 Dec. 2018.
Recommended for acceptance by G. Agrawal.
For information on obtaining reprints of this article, please send e-mail to: reprints@ieee.org, and reference the Digital Object Identifier below.
Digital Object Identifier no. 10.1109/TCC.2016.2552518

users may have excessive resources, resulting in wasted resources. In this study, we investigated an approach for balancing resource utilization in Hadoop systems in heterogenous computing environments such as clouds. In this paper, the term "heterogeneous" means that all nodes have different computing capabilities (e.g., the number of CPUs). In a Hadoop system, the workload of MapReduce jobs submitted by clients typically differs: a job may be split into several tasks that achieve the same function, but involve managing different data blocks. When the default Hadoop job scheduling policy is applied, the task of a single job typically runs on the same *TaskTracker*. When a *TaskTracker* executes the same task for a single job, it is limited by specific resources, despite the other resources remaining idle. For example, assume that jobs are divided into two classes: CPU-bound jobs and I/O-bound jobs. Clients submit the two classes of jobs to Hadoop and then migrate these tasks to the *TaskTracker*. According to the default job scheduling policy of Hadoop, each *TaskTracker* executes the same tasks. However, some *TaskTracker*s are assigned CPU-bound jobs, which are bounded by the CPU resource, and the remaining trackers are assigned I/O-bound jobs, which are bounded by the I/O resource. Tian et al. [23] proposed a dynamic MapReduce (DMR) function for improving resource utilization during various job types. However, the DMR function is applicable in only homogeneous experimental environments. In a heterogeneous environment, the DMR function might not be reasonable for resource utilization. To overcome the limitations of current MapReduce application platforms, this paper first proposes a job scheduler called the *job allocation scheduler* (JAS) for balancing resource utilization in heterogeneous computing environments. The JAS divides jobs into two classes (CPU-bound and I/O-bound) to test the capability of each *TaskTracker* (represented by a capacity ratio). According to the capacity ratio for the two job classes, the *TaskTracker*s are assigned different slots corresponding to the job types to maximize resource utilization. However, if tasks are assigned according to only the two job types, then the JAS may not have the benefit of locality [12], increasing the data transfer time. To address this problem, this paper proposes a modified JAS, called the *job allocation scheduler with locality* (JASL). The JASL can record each node's execution time, and then compare the execution times of the local and non-local nodes to determine whether the task can be executed on the non-local node. In addition, an enhanced JASL, called the *dynamic job allocation scheduler with locality* (DJASL), was developed by adding a dynamic function to the JASL. A performance evaluation was conducted to compare the proposed algorithms with the default job scheduling policy of Hadoop and DMR. The experimental results indicated that the proposed algorithms improved the job scheduling performance of Hadoop and DMR. The JASL (DJASL) enhanced the data locality of Hadoop and the JAS.

The rest of this paper is organized as follows. Section 2 describes the architecture of the Hadoop system and DMR scheduler. In addition, this section reviews related studies. Section 3 presents the components of the proposed method, including job classification and task assignment as well as the JAS, JASL, and DJASL algorithms. Section 4 presents the experimental results obtained in heterogeneous computing environments. Finally, Section 5 concludes the paper.

2 Background

This section introduces Hadoop and related topics. Section 2.1 describes the default scheduler of Hadoop; Section 2.2 introduces job workloads; Section 2.3 presents the problem caused by the default scheduler of Hadoop and job workload; Section 2.4 describes a scheduler designed to address the problem and the problem that it may produce; and Section 2.5 reviews related studies.

2.1 Hadoop Default Scheduler

Hadoop supports the MapReduce programming model originally proposed by Google [9], and it is a convenient approach for developing applications (e.g., parallel computation, job distribution, and fault tolerance). MapReduce comprises two phases. The first phase is the map phase, which is based on a divide-and-conquer strategy. In the divide step, input data are split into several data blocks, the size of which can be set by the user, and are then paralleled by a map task. The second phase is the reduce phase. A map task is executed to generate output data as intermediate data after the map phase is complete, and these intermediate data are then received and the final result is produced.

By default, Hadoop executes scheduling tasks on an FCFS basis, and its execution consists of the following steps:

Step 1 Job submission: When a client submits a MapReduce job to a *JobTracker*, the *JobTracker* adds the job to the *Job Queue*.

Step 2 Job initialization: The *JobTracker* initializes the job in the *Job Queue* by the *JobTracker* by splitting it into numerous tasks; the *JobTracker* then records the data locations of the tasks.

Step 3 Task assignment: When a *TaskTracker* periodically (every 3 seconds by default) sends a *Heartbeat*to a *JobTracker*, the *JobTracker* obtains information on the current state of the *TaskTracker* to determine whether it has available slots. If the *TaskTracker* contains free slots, then the *JobTracker* assigns tasks from the *Job Queue* to the *TaskTracker* according to the number of free slots.

This approach differs considerably from the operating mode implemented in the schedulers of numerous parallel systems [6], [8], [15], [17], [18]. Specifically, some of the task schedulers in such systems, co-scheduler [6] and gang-scheduler [8], operate in heterogeneous environments. Co-schedulers ensure that sub-tasks are initiated simultaneously and are executed at the same pace on a group of workstations, whereas gang-schedulers involve using a set of scheduled threads to execute tasks simultaneously on a set of processors. Other parallel system task schedulers, such as grid-schedulers [15], [17] and dynamic task schedulers [18], have been designed for enhancing performance in scheduling operations involving jobs with dissimilar workloads. The grid-scheduler determines the processing order of jobs assigned to the distribution system, and the dynamic task scheduler operates in environments containing varying system resources and adapts to such variations. This paper proposes a task scheduler that allocates resources at various job workloads in a heterogeneous computing environment.

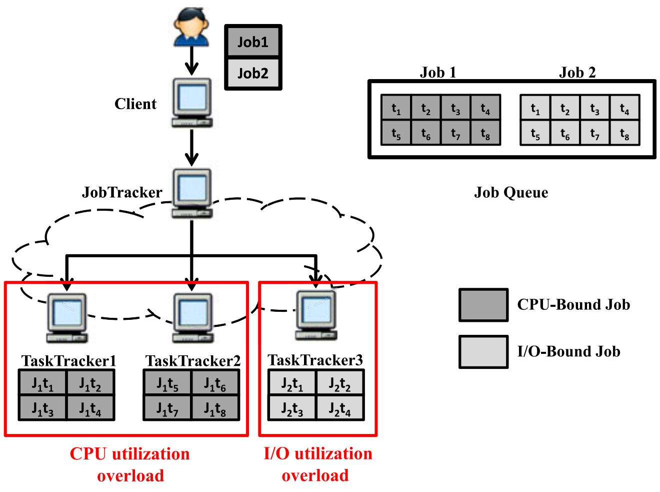


Fig. 1. Imbalanced resource allocation.

2.2 Job Workloads

Rosti et al. [21] proposed that jobs can be classified according to the resources used; some jobs require substantial amount of computational resources, whereas other jobs require numerous I/O resources. In this study, jobs were classified into two categories according to their corresponding workload: 1) CPU-bound jobs and 2) I/O-bound jobs. Characterization and performance comparisons of CPU- and I/O-bound jobs were provided in [16], [20], [26]. CPU- and I/O-bound jobs can be parallelized to balance resource utilization [21], [23].

2.3 Hadoop Problem

As mentioned, Hadoop executes job scheduling tasks on an FCFS basis by default. However, this policy can cause several problems, including imbalanced resource allocation. Consider a situation involving numerous submitted jobs that are split into numerous tasks and assigned to *TaskTrackers*. Executing some of these tasks may require only CPU or I/O resources. Neglecting the workloads of a job may lead to imbalanced resource allocation. Fig. 1 illustrates an imbalanced resource allocation scenario in the Hadoop default scheduler.

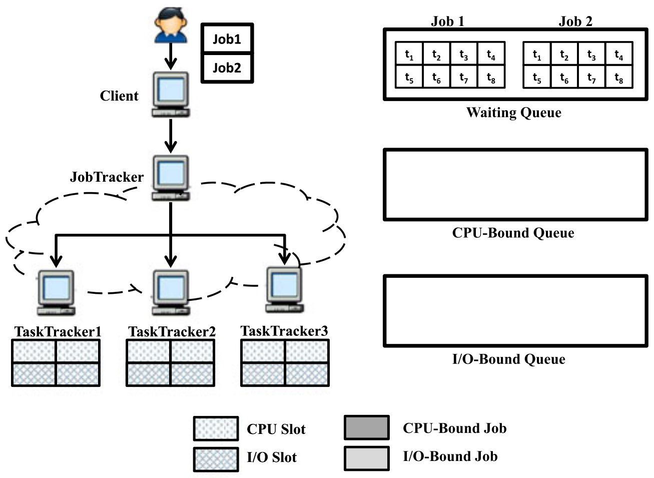
Assume that the *Job Queue* contains two jobs: a CPU-bound job (i.e., *Job*1) and an I/O-bound job (i.e., *Job*2). *Job*1 and *Job*2 are initialized with eight map tasks. Moreover, *TaskTracker*1, *TaskTracker*2, and *TaskTracker*3 send sequential *Heartbeat* messages to the *JobTracker*. According to the default job scheduler of Hadoop, the *JobTracker* submits tasks to the *TaskTracker* according to the *Heartbeat* order. *TaskTracker*1 and *TaskTracker*2 are assumed to execute CPU-bound jobs; therefore, they have high CPU utilization and low I/O utilization, causing them to be bounded by CPU resources. By contrast, *TaskTracker*3 is assigned to execute I/O-bound jobs; therefore, *TaskTracker*3 has high I/O utilization but low CPU utilization, causing it to be bounded by I/O resources. Because the default job scheduler in Hadoop does not balance resource utilization, some tasks in the *TaskTracker* cannot be completed until resources used to execute other tasks are released. Because some tasks must wait for resources to be released, the task execution time is prolonged, leading to poor performance.

2.4 Dynamic Map-Reduce Scheduler

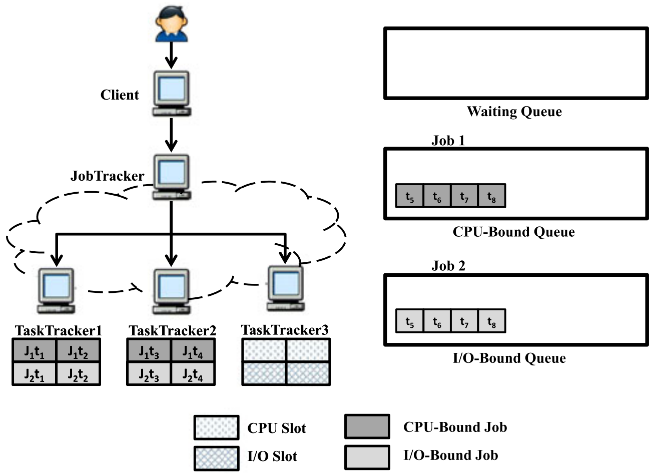
To address the imbalanced resource allocation problem of the default scheduler in Hadoop, as described in Section 2.3, Tian et al. [23] proposed a balanced resource utilization algorithm (DMR) for balancing CPU- and I/O-bound jobs. They proposed a classification-based triple-queue scheduler to determine the category of one job and then parallelize various job types and thus balance the resources of *JobTrackers* by using CPU- and I/O-bound queues.

Consider the example shown in Fig. 2. The DMR scheduler first evenly divides the slots into CPU and I/O slots. When *Job*1 and *Job*2 are added to the *Waiting Queue*, they are divided into eight map tasks (Fig. 2a). After the scheduler determines the workload types of *Job*1 and *Job*2, the jobs are added to the CPU- or I/O-bound queue. *TaskTracker*1, *TaskTracker*2, and *TaskTracker*3 send sequential *Heartbeat* messages to the *JobTracker*. The *JobTracker* then determines that all three *TaskTrackers* have two idle CPU slots and two idle I/O slots according to the *Heartbeat* information (assuming that each *TaskTracker* has four slots). The *JobTracker* then queries the *Job Queue* to determine whether any task can be assigned to the TaskTrackers. The *JobTracker* knows that the *CPU-bound queue* contains *Job*1 and that the *I/O-bound queue* contains *Job*2. It then assigns two CPU-bound job tasks, J_1t_1 and J_1t_2, and two I/O-bound job tasks, J_2t_1 and J_2t_2, to *TaskTracker*1 (Fig. 2b). *TaskTracker*2 and *TaskTracker*3 execute similar steps to those of *TaskTracker*1. These steps are repeated until all jobs are completed. According to the DMR concept, each *TaskTracker* has a CPU-bound job and I/O-bound job. Furthermore, this approach demonstrates improved performance because each *TaskTracker* improves overall resource utilization (Fig. 2c).

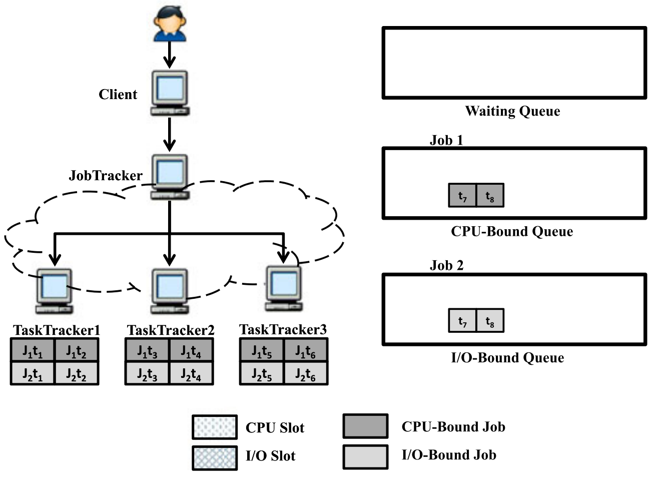
The DMR approach generally improves resource utilization in a Hadoop system. However, in a heterogeneous computation environment, the DMR approach may cause uneven resource utilization in some *TaskTrackers*, thus reducing the performance of the Hadoop system. For example, assume that three *TaskTrackers*, *TaskTracker*1, *TaskTracker*2, and *TaskTracker*3, have distinct capabilities (Fig. 3). *TaskTracker*1 can execute two CPU-bound jobs and one I/O-bound job simultaneously; therefore, this TaskTracker has two CPU slots and one I/O slot. *TaskTracker*2 can execute one CPU-bound job and two I/O-bound jobs simultaneously (i.e., *TaskTracker*2 contains one CPU slot and two I/O slots). *TaskTracker*3 can execute one CPU-bound job and three I/O-bound jobs simultaneously (i.e., *TaskTracker*3 has three CPU slots and one I/O slot). Nevertheless, according to the DMR approach, each *TaskTracker* has two CPU slots and two I/O slots (implying a total of four slots). After receiving jobs from clients, the *JobTracker* assigns the tasks to a *TaskTracker*. Each *TaskTracker* contains two CPU-bound tasks and two I/O-bound tasks. Therefore, *TaskTracker*1 has one I/O-bound task that must wait for the I/O resources to be released, resulting in its I/O capacity becoming overloaded. *TaskTracker*2 has one CPU slot that must wait for CPU resources to be released; therefore, the CPU capacity of *TaskTracker*2 becomes overloaded. Finally, *TaskTracker*3 has one CPU slot that must wait for CPU resources to be released; therefore, the CPU capacity of *TaskTracker*3 becomes overloaded. Furthermore, *TaskTracker*3 includes one idle I/O slot, indicating



(a) When a client submits a new job, the submitted job is added to the waiting queue. Subsequently, the scheduler classifies the job type.



(b) After the scheduler classifies the job type, the jobs are added to the CPU-bound queue or the I/O-bound queue. The *JobTracker* then assigns these tasks according to the number of free CPU or I/O slots contained in the *TaskTracker*.



(c) The *JobTracker* assigns tasks until all of the *TaskTrackers* have no free slots.

Fig. 2. Workflow of a DMR scheduler.

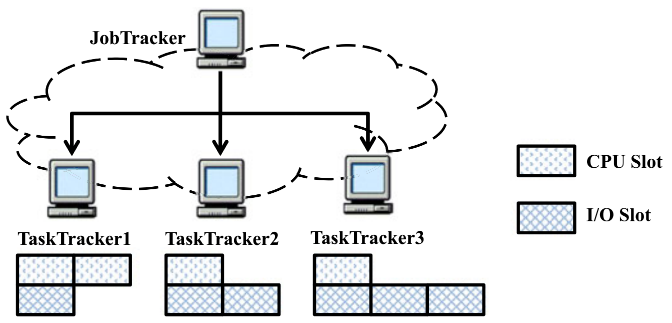


Fig. 3. Different capabilities of *TaskTrackers*.

that its I/O resources are not effectively used. According to this example, the DMR may exhibit poor performance in a heterogeneous environment because of its inefficient resource utilization. Therefore, resource allocation is a critical concern in heterogeneous computing environments involving varying job workloads.

2.5 Related Studies

Because of the increase in the amount of data, balancing resource utilization is a valuable method for improving the performance of a Hadoop system.[1] Numerous resource allocation algorithms have been proposed in recent years.

Bezerra et al. [7] a RAMDISK for temporary storage of intermediate data. RAMDISK has high throughput and low latency and this allows quick access to the intermediate data relieving in the hard disk. Thus, adding RAMDISK improves the performance of the shared input policy. Ghoshal et al. [10] a set of pipelining strategies to effectively utilize provisioned cloud resources. The experiments on the ExoGENI cloud testbed demonstrates the effectiveness of our approach in increasing performance and reducing failures.

Isard et al. [14] mapped a resource allocation problem to a graph data structure and then used a standard solver, called *Quincy*, for computing the optimal online scheduler. When fairness is required, Quincy increases fairness, substantially improving data locality.

Tumanov et al. [24] focused on the resource allocation of mix workloads in heterogeneous clouds and proposed an algebraic scheduling policy called *Alsched* for allocating resources to mixed workloads in heterogeneous clouds. Alsched allocates resources by considering the customizable utility functions submitted as resource requests.

Schwarzkopf et al. [22] presented a novel approach to address the increasing scale and the need for a rapid response to changing requirements. These factors restrict the rate at which new features can be deployed and eventually limit cluster growth. Two schedulers, an monolithic scheduler and a statically partitioned scheduler, were presented in [22] to achieve flexibility for large computing clusters, revealing that optimistic concurrency over a shared state is a viable and attractive approach for cluster scheduling.

Max-min fairness is a resource allocation mechanism used frequently in data center schedulers. However,

1. This study applied a different approach by proposing a job scheduling algorithm to achieve this goal.

numerous jobs are limited to specific hardware or software in the machines that execute them. Ghodsi et al. [11] proposed an off line resource allocation algorithm called *Constrained Max-Min Fairness*, which is an extension of max-min fairness and supports placement constraints. They also proposed an on-line version called *Choosy*. Apache released a new version of the Hadoop system, called *YARN* or *MapReduce2.0* (MRv2)[2], which is currently adopted only by Yahoo. The two major function of *JobTracker*s, namely resource management and job scheduling and monitoring, are split into two components called *ResourceManager* and *NodeManager*. *ResourceManager* is used to schedule the demand of resources that applications require, and *NodeManager* is used to monitor the use of resources, such as CPU, memory, disk, and network resources, by running applications. However, default schedulers of both Hadoop and YARN still do not support heterogenous computing environments. The main focus of YARN is on improving resource utilization and load balancing. Not only resource utilization and load balancing are the important issue of cloud computing but data locality also the problem we need to concern. Otherwise, although there are many different kinds of mechanism and method to deal with job scheduling problem, but the kinds of mixed workloads and diverse of heterogeneous environments are important factor to effect the results. Therefore, the novel scheduling algorithms and policies proposed in this paper are incomparable and take advantage of related studies for some reason on both performance and data locality and also applicable to both Hadoop and YARN on these problems.

3 Proposed Algorithms

This section presents the proposed JAS algorithm, which provides each *TaskTracker* with a distinct number of slots according to the ability of the *TaskTracker* in a heterogeneous environment.

3.1 JAS Algorithm

When a *TaskTracker* sends a *Heartbeat* message, the following phases of the JAS algorithm are executed.

Step 1: Job classification: When jobs are in the *Job Queue*, the *JobTracker* cannot determine the job types (i.e., CPU-bound or I/O-bound). Thus, the job types must be classified and the jobs must be added to the corresponding queue according to the method introduced in Section 3.1.1.

Step 2: *TaskTracker* slot setting: After a job type is determined, the *JobTracker* must assign tasks to each *TaskTracker* depending on the number of available slots for each type in each *TaskTracker* (CPU slots and I/O slots). Thus, the number of slots must be set on the basis of the individual ability of each *TaskTracker* according to the methods introduced in Sections 3.1.2 and 3.1.3.

Step 3: Tasks assignment: When a *TaskTracker* sends a *Heartbeat* message, the *JobTracker* receives the numbers of idle CPU slots and I/O slots and then assigns various types of job tasks to the corresponding slots for processing (i.e., the tasks of a CPU-bound job are assigned to CPU slots, and vice versa). Sections 3.1.4 and 3.1.5 introduce the task assignment procedures.

TABLE 1
Parameters of Job Classification

Notation	Meaning
n	number of map tasks
MID	Map Input Data
MOD	Map Output Data
SID	Shuffle Input Data
SOD	Shuffle Output Data
$MTCT$	Map Task Completed Time
$DIOR$	Disk Average I/O Rate

3.1.1 Job Classification

Algorithm 1 presents job classification process. When a *TaskTracker* sends a *Heartbeat* message, the *JobTracker* can determine the number of tasks that have been executed in the *TaskTracker*, which enables it to determine the states of these tasks. When these tasks are complete, the *JobTracker* can receive information on the tasks (Table 1).

The parameters used to classify jobs (Table 1) were detailed in [23]. Assume that a *TaskTracker* has n slots; the *TaskTracker* executes the same n map tasks, and the completion times of the map tasks are identical. A map task generates data throughput, including map input data (MID), map output data (MOD), shuffle output data (SOD), and shuffle input data (SID). Hence, n map tasks generate the total data $throughput = n * (MID + MOD + SOD + SID)$, and the amount of data that can be generated from a *TaskTracker* in 1 s is

$$throughput = \frac{n * (MID + MOD + SOD + SID)}{MTCT}, \quad (1)$$

where $MTCT$ is the map task completion time.

Algorithm 1. JOB_CLASSIFICATION (*Heartbeat*)

```
1  Obtain TaskTracerQueues information from Heartbeat:
2  for $task$ in $TaskTracker$ do
3      if $task$ has been completed by $TaskTracker$ then
4          obtain the task information from TaskTracker;
5          compute $throughput := \frac{n*(MID+MOD+SOD+SID)}{MTCT}$;
6          if task belongs to a job $J$ that has not been classified then
7              if $throughput < DIOR$ then
8                  set $J$ as a CPU-bound job;
9                  move $J$ to the CPU Queue;
10             else
11                 set $J$ as an I/O-bound job;
12                 move $J$ to the I/O Queue;
13     if task belongs to a CPU-bound job then
14         record the execution time of the task on TaskTrackerCPUCapability;
15     else
16         record the execution time of the $task$ on TaskTrackerIOCapability.
```

TABLE 2
Parameters Used for Setting CPU Slots

Notation	Meaning
$T = \{t_1, t_2, \ldots, t_{n-1}, t_n\}, \lvert T\rvert = n$	the set of tasks on a CPU-bound job
ET_{t_i}	the execution time of task t_i, where $t_i \in T$
$M_i = \{t_j \in T \mid t_j$ runs on $TaskTracker_i\}$	the set of tasks run on $TaskTracker_i$
$e_i = \sum_{t_j \in M_i} g_{t_j}$	the total execution time of the tasks run on $TaskTracker_i$
$r_j, j = 1, \ldots, m$	the label of $TaskTracker_j$
s	the number of all slots on Hadoop
c_y	the CPU execution capability of the $TaskTracker_y$
k_y	the number of CPU slots in $TaskTracker_y$

The *JobTracker* can determine the amount of data that a single map task can generate from a *TaskTracker* in 1 s (i.e., *throughput*). When the throughput is less than the disk average I/O rate (DIOR), the *JobTracker* classifies a job according to the determined information; in this case, the total data throughput generated by n map tasks is still less than the average I/O read/write rate. When the map tasks require few I/O resources (i.e., $throughput < DIOR$), the *JobTracker* classifies the job as CPU-bound. Otherwise, $throughput \geq DIOR$, implying that the total data throughput generated by n map tasks is at least equal to the average I/O read/write rate. In this case, the *JobTracker* classifies the job as I/O-bound. After all jobs are classified, the *JobTracker* must record the execution time of all tasks, which can be used to compute the number of CPU slots (I/O slots) of each *TaskTracker*.

3.1.2 CPU Slot Setting

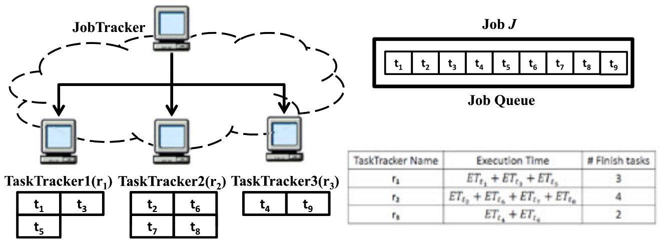
Algorithm 2 presents the procedure for setting the CPU slots of a *TaskTracker*, and Table 2 lists the parameters used in these procedures

Algorithm 2. SET_CPU_SLOT(*Job Queue*)

```
1  for Job in Job Queue do
2      if Job has been completed and Job is CPU-bound then
3          obtain the task information from TaskTracker;
4          compute the TaskTracker capability according to TaskTrackerCPUCapability;
5          $c_y := \frac{|M_y|}{e_y}$;
6          for each TaskTracker do
7              $k_y :=$ (the number of CPU slots) $* \frac{c_y}{\sum_{j=1}^{m} c_j}$;
8              record $k_y$ on TaskTrackerCPUTable;
9              SetTaskTrackerCPUTable := 1;
10         return TaskTrackerCPUslot according to TaskTrackerCPUTable;
11         break.
```

$$c_y = \frac{|M_y|}{e_y}; \tag{2}$$

$$k_y = \text{the number of CPU slots} * \frac{c_y}{\sum_{j=1}^{m} c_j}. \tag{3}$$



TaskTracker Name	Execution Time	# Finish tasks
r_1	$ET_{t_1} + ET_{t_3} + ET_{t_5}$	3
r_2	$ET_{t_2} + ET_{t_6} + ET_{t_7} + ET_{t_8}$	4
r_3	$ET_{t_4} + ET_{t_9}$	2

Fig. 4. Illustration of Algorithm 2.

In Algorithm 1, when a task belonging to a CPU-bound job has been completed, the *JobTracker* records the execution time of the task in *TaskTrackerCPUCapability*. When the *JobTracker* detects that the number of CPU slots in the *TaskTracker* has not been set, it executes Algorithm 2 to set the number of CPU slots. In Algorithm 2, the *JobTracker* reads the execution time of each task in *TaskTrackerCPUCapability* and computes the CPU capability of each *TaskTracker* according to (2). Finally, when the *JobTracker* computes the CPU capability ratio of each *TaskTracker* according to (3), it can determine the number of CPU slots in each *TaskTracker*.

Fig. 4 illustrates how the number of CPU slots in each *TaskTracker* is determined. Assume that a client submits a job J to the *JobTracker*, which is a CPU-bound job divided into nine tasks. This Hadoop system contains three *TaskTracker*s: r_1, r_2, and r_3. The *JobTracker* distributes the nine tasks such that t_1, t_3, and t_5 are assigned to r_1; t_2, t_6, t_7, and t_8 are assigned to r_2; and t_4 and t_9 are assigned to r_3. After a task has been completed, the *JobTracker* classifies the task as a CPU-bound job and then records the execution time of that task in *TaskTrackerCPUCapability*. For example, when t_1 has been completed and then the *JobTracker* classifies it as a CPU-bound job, the *JobTracker* records its execution time in r_1 in *TaskTrackerCPUCapability*, therefore, *TaskTrackerCPUCapability* contains the record that t_1 has been completed by r_1. These steps are repeated for recording t_2–t_9 in *TaskTrackerCPUCapability*. Fig. 4 illustrates the final results regarding *TaskTrackerCPUCapability*. After a job is complete, the *JobTracker* determines whether the CPU slot in a *TaskTracker* has been set to 1. If this is the case, then the *JobTracker* skips Algorithm 2; otherwise, the *JobTracker* executes Algorithm 2. According to *TaskTrackerCPUCapability*, the *JobTracker* computes the CPU capability of each *TaskTracker* according to (2). Equation (2) shows the number of tasks belonging to J assigned to a *TaskTracker* that can be completed in 1 s. Let s be the number of slots in the Hadoop system. For example, the capacity of r_1 is $c_1 = \frac{t_1+t_3+t_5}{3}$; the capacity of r_2 is $c_2 = \frac{t_2+t_6+t_7+t_8}{4}$; and the capacity of r_3 is $c_3 = \frac{t_4+t_9}{2}$. After determining the capacity of each *TaskTracker*, the *JobTracker* uses (3) to compute the CPU capability ratio of each *TaskTracker* and calculates the number of CPU slots in each *TaskTracker*. For example, the number of CPU slots in r_1 is $k_1 = \frac{s}{2} * \frac{c_1}{c_1+c_2+c_3}$; the number of CPU slots in r_2 is $k_2 = \frac{s}{2} * \frac{c_2}{c_1+c_2+c_3}$; and the number of CPU slots in r_3 is $k_3 = \frac{s}{2} * \frac{c_3}{c_1+c_2+c_3}$. Because the number of CPU slots (number of I/O slots) is equal to half the number of Hadoop slots, the number of CPU slots (I/O slots) is set to $\frac{s}{2}$.

After executing Algorithm 2, the *JobTracker* can determine the number CPU slots in each *TaskTracker*. This is useful for improving each *TaskTracker*'s CPU resource utilization.

TABLE 3
Parameters Used for Setting I/O Slots

Notation	Meaning
$L = \{t_1, t_2, \ldots, t_n\}, \lvert L\rvert = n$	the set of tasks on a I/O-bound job
l_{t_i}	the execution time of task t_i, where $t_i \in L$
$N_i = \{t_j \in L \mid t_j$ run on $TaskTracker_i\}$	the set of tasks run on $TaskTracker_i$
$f_i = \sum_{t_j \in N_i} l_{t_j}$	the total execution time of the tasks run on $TaskTracker_i$
d_y	the I/O execution capability of $TaskTracker_y$
i_y	the number of I/O slots in $TaskTracker_y$

3.1.3 I/O Slot Setting

A *TaskTracker* executes Algorithm 3 to set I/O slots. Table 3 lists the parameters used in the algorithm.

Suppose that m $TaskTracker$s exist: $TaskTracker_1, TaskTracker_2, \ldots, TaskTracker_m$. Define

$$d_y = \frac{n_y}{f_y}, \tag{4}$$

and

$$i_y = (\text{the number of I/O slots}) * \frac{d_y}{\sum_{j=1}^{m} d_j}. \tag{5}$$

In Algorithm 1, when a task belonging to an I/O-bound job has been completed, the *JobTracker* records the task execution time in *TaskTrackerIOCapability*. If the *JobTracker* detects that the number of I/O slots in a *TaskTracker* has not been set, it executes Algorithm 3 to set the number of I/O slots. In Algorithm 3, the *JobTracker* reads each task's execution time in *TaskTrackerIOCapability* and then computes each *TaskTracker*'s I/O capability according to (4). Finally, the *JobTracker* computes each *TaskTracker*'s I/O capability ratio according to (5) and then sets the number of I/O slots for each *TaskTracker*.

Algorithm 3. SET_I/O_SLOT(*Job Queue*)

```
1  for Job in Job Queue do
2      if Job has been finished and Job is I/O-bound then
3          obtain the task information from TaskTracker;
4          compute the TaskTracker capability according to TaskTrackerIOCapability;
5          $d_y := \frac{n_y}{f_y}$;
6          for each TaskTracker do
7              $i_y :=$ (the number of I/O slots) $* \frac{d_y}{\sum_{j=1}^{m} d_j}$;
8              record $i_y$ on TaskTrackerIOTable;
9              SetTaskTrackerIOTable := 1;
10         return TaskTrackerIOslot according to TaskTrackerIO Table;
11         break.
```

3.1.4 CPU Task Assignment

The *JobTracker* executes Algorithm 1 to classify each job and then executes Algorithm 2 to set the number of CPU slots for each *TaskTracker*. According to *Heartbeat* information, the *JobTracker* determines the number of tasks (belonging to a CPU-bound job) executed for a *TaskTracker* and calculates the number of idle CPU slots (recorded in *AvailableCPUSlot*) in the *TaskTracker*. After the *JobTracker* obtains the *AvailableCPUSlot* information of the *TaskTracker*, the *JobTracker* executes Algorithm 4 to assign CPU tasks to each *TaskTracker*.

Algorithm 4. CPU_TASK_ASSIGN

```
1  for each AvailableCPUSlot do
2      for Job in the Job Queue do
3          if Job has not been classified then
4              select a task of Job and move it to the TaskTrackerQueues of a TaskTracker according to the Heartbeat information;
5              break;
6      if ∃ a task has not been selected and moved to the TaskTrackerQueues of a TaskTracker then
7          for Job in the CPU Queue do
8              if Job has not been completed then
9                  select a task of Job and move it to the TaskTrackerQueues of the corresponding TaskTracker according to the information of Heartbeat;
10                 break;
11         if ∃ a task that has not been selected and moved it to the TaskTracker's TaskTrackerQueues then
12             for Job in I/O Queue do
13                 if Job has not been finished then
14                     select a task of Job and move it to the TaskTrackerQueues of the TaskTracker according to Heartbeat information;
15                     break.
```

For each *AvailableCPUSlot*, the *JobTracker* first queries the *Job Queue*. If an unclassified job appears in the *Job Queue*, then the *JobTracker* selects one task from the job and assigns it to a *TaskTracker*, terminating the iteration. Subsequently, *AvailableCPUSlot* is reduced by one and the *JobTracker* initiates the next iteration. The *JobTracker* then requeries the *Job Queue* until all jobs in the *Job Queue* have been classified. The *JobTracker* then queries the *CPU Queue*, and if it detects an unfinished job in this queue, the *JobTracker* assigns a task from the unfinished jobs in the *CPU Queue* to a *TaskTracker*, terminating the iteration. Subsequently, *AvailableCPUSlot* is reduced by one and the *JobTracker* initiates the next iteration. The *JobTracker* requeries the *CPU Queue* until no unfinished jobs. These steps are repeated until *AvailableCPUSlot* is zero, meaning that the *TaskTracker* has no idle CPU slots; hence, the *JobTracker* terminates the execution of Algorithm 4.

If *AvailableCPUSlot* is not zero and the *JobTracker* does not select any task from the *Job Queue* or the *CPU Queue* for the *TaskTracker*, then the *JobTracker* queries the *I/O Queue* to ensure that no idle CPU slots are wasted. If an unfinished job is detected in the *I/O Queue*, the *JobTracker* assigns a task from this queue to a *TaskTracker*, terminating the iteration. These steps are repeated until no unfinished jobs remain in the *I/O Queue* or *AvailableCPUSlot* is zero. The *JobTracker* then terminates Algorithm 4. Thus, the proposed Algorithm 4 enables

TABLE 4
JAS Parameters

Notation	Meaning
SetTaskTrackerCPUTable	a Boolean variable indicating whether the CPU slot of each TaskTracker has been set by the JobTracker
SetTaskTrackerIOTable	a Boolean variable indicating whether the I/O slot of each TaskTracker has been set by the JobTracker
TaskTrackerCPUslot	the number of CPU slots in the TaskTracker
TaskTrackerIOslot	the number of I/O slots in the TaskTracker
TaskTrackerRunningCPUtask	the number of CPU-bound job tasks currently being executed by the corresponding TaskTracker
TaskTrackerRunningIO	the number of I/O-bound job tasks currently being executed by the corresponding TaskTracker
AvailableCPUSlots	the number of idle CPU slots in a TaskTracker (these idle CPU slots can work with CPU-bound job tasks)
AvailableIOSlots	TaskTrackerIOslot − TaskTrackerRunningIO

the *JobTracker* to allocate tasks accurately to the idle CPU slots of *TaskTracker*s, thus preventing wastage of idle CPU slots.

3.1.5 I/O Task Assignment

The *JobTracker* executes Algorithm 1 to classify each job and Algorithm 3 to set the number of I/O slots for each *TaskTracker*. On the basis of *Heartbeat* information, the *JobTracker* determines the number of tasks (belonging to an I/O-bound job) executed by a *TaskTracker*, and then calculates the number of idle I/O slots (recorded in *AvailableIOSlot*) existing in this Tracker. After the JobTracker obtains the *AvailableIOSlot* of the *TaskTracker*, it executes Algorithm 5 to assign I/O tasks to each *TaskTracker*.

Algorithm 5. I/O_TASK_ASSIGN

```
1  for each AvailableIOSlot do
2      for Job in the Job Queue do
3          if Job has not been classified then
4              select a task of Job and move it to the Task-
               TrackerQueues of a TaskTracker according to
               the Heartbeat information;
5              break;
6      if ∃ a task is not selected and moved to the TaskTracker-
       Queues of a TaskTracker then
7          for Job in the I/O Queue do
8    if Job has not been completed then
9              select a task of Job and move it to the Task-
               TrackerQueues of the TaskTracker according to
               the Heartbeat information;
10             break;
11         if ∃ a task that is not selected and moved it to
           TaskTracker's TaskTrackerQueues then
12         for Job in the CPU Queue do
13             if Job has not been finished then
14                 select a task of Job and move it to Task-
                   TrackerQueues of the corresponding Task-
                   Tracker according to the information of
                   Heartbeat;
15                 break.
```

For each *AvailableIOSlot*, the *JobTracker* first queries the *Job Queue*. If an unclassified job is detected in the *Job Queue*, the *JobTracker* selects one task from the job and moves it to a *TaskTracker*, terminating the iteration. Subsequently, *AvailableIOSlot* is reduced by one and the *JobTracker* initiates the next iteration. The *JobTracker* requeries the *Job Queue* until no unclassified jobs remain in the queue. The *JobTracker* then queries the *I/O Queue*, if an unfinished job exists in the *I/O Queue*, the *JobTracker* assigns a task from the unfinished jobs in the queue to a *TaskTracker*, terminating the iteration. Subsequently, *AvailableIOSlot* is reduced by one and the *JobTracker* begins the next iteration. The *JobTracker* requeries the *I/O Queue* until no unfinished jobs remain in the queue. These steps are repeated until *AvailableIOSlot* is zero, meaning that the *TaskTracker* has no idle I/O slots; hence, the *JobTracker* terminates Algorithm 5.

If *AvailableIOSlot* is not equal to zero and the *JobTracker* does not assign any task from the *Job Queue* or *I/O Queue* to a *TaskTracker*, the *JobTracker* queries the *CPU Queue* to ensure that no idle I/O slots are wasted. If an unfinished job exists in the *CPU Queue*, the *JobTracker* assigns a task from this queue to a *TaskTracker*, terminating the iteration. These steps are repeated until no unfinished jobs remain in the *CPU Queue* or *AvailableIOSlot* is zero. The *JobTracker* then terminates Algorithm 5. Therefore, the proposed Algorithm 5 enables the *JobTracker* to allocate tasks accurately to the idle I/O slots of *TaskTracker*s, thus preventing the waste of idle I/O slots.

3.1.6 Complete JAS Algorithm

All the algorithms presented in the preceding sections were compiled into one algorithm, forming Algorithm 6. Table 4 shows the required parameters.

Slots are set according to each *TaskTracker*'s ability, and tasks from either the CPU- or I/O-bound queue are assigned to *TaskTracker*s. The proposed JAS algorithm reduces the execution time compared with the FCFS scheduling policy of Hadoop.

3.2 Improving the Data Locality of the JAS Algorithm

3.2.1 Problems of the JAS Algorithm

Although Hadoop tends to assign tasks to the nearest node possessing its block, the JAS algorithm assigns tasks according to the number of CPU and I/O slots in *Tasktracker*s. Because of the property of the JAS, data locality is lost and a substantial amount of network traffic occurs. Fig. 5 illustrates these problems.

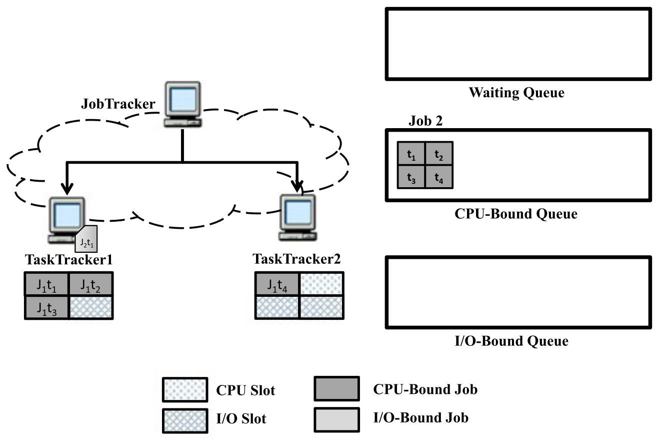
Assume that *TaskTracker*s are available (i.e., *TaskTracker*1 and *TaskTracker*2); *TaskTracker*1 has three CPU slots and one I/O slot, and *TaskTracker*2 contains two CPU slots and two I/O slots. Job 1 and Job 2 are CPU-bound jobs. According to Algorithm 6, the *JobTracker* assigns J_1t_1, J_1t_2, and J_1t_3 to *TaskTracker*1, and assigns J_1t_4 to *TaskTracker*2 (Fig. 5a). Job 2 remains in the CPU-bound queue; subsequently, the *JobTracker* assigns its tasks. When the JAS is applied, the *JobTracker* tends to assign J_2t_1 to *TaskTracker*2. However, because the data block required by J_2t_1 is in *TaskTracker*1; *TaskTracker*2 must retrieve this block from *TaskTracker*1. The transfer of the data block results in extra network traffic, eliminating the benefit of data locality (Fig. 5b).

Algorithm 6. Job_Allocation_Scheduler (JAS)

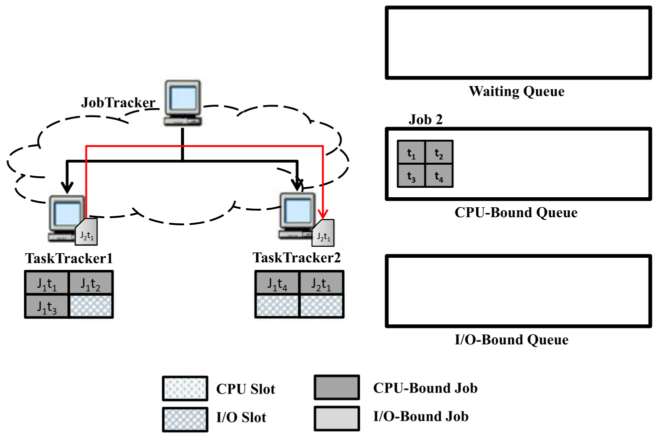
```
1  When a batch of jobs are submitted into JobTracker:
2  add jobs into Job Queue;
3  SetTaskTrackerCPUTable := 0;
4  SetTaskTrackerIOTable := 0;
5  while receive Heartbeat by TaskTracker do
6      TaskTrackerCPUslot := 0;
7      TaskTrackerIOslot := 0;
8      obtain TaskTrackerRunningCPUtask from Heartbeat information;
9      obtain TaskTrackerRunningIOtask from Heartbeat information;
10     AvailableCPUSlots := 0;
11     AvailableIOSlots := 0;
12     JOB_CLASSIFICATION(Heartbeat);
13     if SetTaskTrackerCPUTable == 1 then
14         obtain TaskTrackerCPUslot according to TaskTrackerCPUTable;
15     else
16         TaskTrackerCPUslot:=SET_CPU_SLOT (Job Queue);
17     if SetTaskTrackerIOTable == 1 then
18         obtain TaskTrackerIOslot according to TaskTrackerIOTable;
19     else
20         TaskTrackerIOslot:=SET_IO_SLOT(Job Queue);
21     if TaskTrackerCPUslot == 0 then
22         TaskTrackerCPUslot := default CPU slot;
23     if TaskTrackerIOslot == 0 then
24         TaskTrackerIOslot := default I/O slot;
25     AvailableCPUSlots := TaskTrackerCPUslot − TaskTracker RunningCPUtask;
26     AvailableIOSlots := TaskTrackerIOslot− TaskTracker RunningIOtask;
27     CPU_TASK_ASSIGN(AvailableCPUSlot);
28     IO_TASK_ASSIGN(AvailableIOSlot).
```

3.2.2 Improving the JAS

To address the problem described in Section 3.2.1, the JAS algorithm was modified to ensure that data locality is retained. First, when Algorithm 1 is executed, the execution times of all *TaskTrackers* in *LocalityBenefitTable* in the cluster, including the times required by the *TaskTrackers* to execute local and non-local map-tasks, are recorded as indicated by Algorithm 7. The *JobTracker* then assigns either CPU- or I/O-bound tasks according to (6). If *NonLocalBenefit* is true, then the *TaskTracker* can execute a non-local task; otherwise, it can execute only local tasks, as indicated by Algorithm 8



(a) *JobTracker* assign tasks of Job 1 according to *TaskTracker*s CPU slots. Job 2 is in the CPU-bound queue waiting for the assignment of the *JobTracker*.



(b) According to Algorithm 6, the *JobTracker* assigns J_2t_1 to *TaskTracker*2. Because the data block required by J_2t_1 is in *JobTracker*1, it needs to be moved from *TaskTracker*1 to *TaskTracker*2, which may increase network traffic.

Fig. 5. JAS algorithm may increase network traffic.

$$NonLocalBenefit = \begin{cases} true & \text{if } ET_i > ET_j + Transfer\_time; \\ false & \text{else.} \end{cases} \quad (6)$$

Let ET_i and ET_j be the execution times of two *TaskTrackers* in a cluster, and let $Transfer\_time$ be the transfer time of a block from $TaskTracker_i$ to $TaskTracker_j$.

To address the locality problem, Algorithm 5 can be modified in a similar manner as Algorithm 4, deriving Algorithm 8. Algorithm 9 presents the enhanced JASL. As demonstrated in Section 4, Algorithm 9 exhibits superior performance to the Hadoop scheduling policy and greater data locality than does the JAS algorithm.

Algorithm 7. JOB_CLASSIFICATION_L (*Heartbeat*)

```
1  obtain TaskTracerQueues information from Heartbeat:
2  Initialize LocalityBenifitTable: = 0;
3  for task in TaskTracker do
4      if task has been completed by TaskTracker then
5          obtain the task information from TaskTracker;
6          compute $throughput = \frac{n*(MID+MOD+SOD+SID)}{MTCT}$;
7          if task belongs to a job J that has not been classified then
8              if $result < DIOR$ then
9                  set J as a CPU-bound job;
10                 move J to CPU Queue;
11             else
12                 set J as a IO-bound job;
13                 move J to IO Queue;
14     if task belongs to a CPU-bound job then
15         record the execution time of task on TaskTrackerCPUCapability;
16     else
17         record the execution time of task on TaskTrackerIOCapability.
18     record the execution time of task on LocalityBenefitTable;
```

Algorithm 8. CPU_TASK_ASSIGN_L

```
1  for each AvailableCPUSlot do
2      $ET_1$: the execution time of the task executed by the current TaskTracker;
3      $ET_2$: the execution time of the task executed by the remote TaskTracker;
4      Transfer_Time: the transfer time between the TaskTrackers;
5      obtain $ET_1$ and $ET_2$ + Transfer_Time from LocalityBenefitTable;
6      for Job in Job Queue do
7          if Job has not been classified then
8              select a task of Job and move it to Task TracerQueues of a TaskTracker according to the Heartbeat information;
9              break;
10     if ∃ a task is not selected and moved to the TaskTracerQueues of a TaskTracker then
11         for Job in CPU Queue do
12             if Job has not been completed then
13                 if ∃ tasks belonging to local tasks then
14                 assign one of them to TaskTracker;
15                 break;
16             else
17                 if NonLocalBenefit==true then
18                     assign a non-local task;
19                     break;
20                 break;
21         if ∃ a task is not selected and moved to TaskTracerQueues of a TaskTracker then
22             for Job in I/O Queue do
23                 if Job has not been completed then
24                     select a task of Job and move it to TaskTrackerQueues of the TaskTracker according to the Heartbeat information;
25                     break.
```

Algorithm 9. Job_Allocation_Scheduler_with_Locality (JASL)

```
1  When a batch of jobs is submitted to JobTracker:
2  add jobs into Job Queue;
3  SetTaskTrackerCPUTable := 0;
4  SetTaskTrackerIOTable := 0;
5  Initialize LocalityBenifitTable := 0;
6  while receive Heartbeat by TaskTracker do
7      TaskTrackerCPUslot := 0;
8      TaskTrackerIOslot := 0;
9      obtain TaskTrackerRunningCPUtask from Heartbeat information;
10     obtain TaskTrackerRunningIOtask from Heartbeat information;
11     AvailableCPUSlots := 0;
12     AvailableIOSlots := 0;
13     JOB_CLASSIFICATION_L(Heartbeat);
14     if SetTaskTrackerCPUTable == 1 then
15         obtain TaskTrackerCPUslot according to TaskTrackerCPUTable;
16     else
17         TaskTrackerCPUslot:=SET_CPU_SLOT(Job Queue);
18     if SetTaskTrackerIOTable == 1 then
19         get TaskTrackerIOslot according to TaskTrackerIO Table;
20     else
21         TaskTrackerIOslot := SET_IO_SLOT(Job Queue);
22     if TaskTrackerCPUslot == 0 then
23         TaskTrackerCPUslot := default CPU slot;
24     if TaskTrackerIOslot == 0 then
25         TaskTrackerIOslot := default I/O slot;
26     AvailableCPUSlots := TaskTrackerCPUslot − TaskTracker RunningCPUtask;
27     AvailableIOSlots:= TaskTrackerIOslot − TaskTracker RunningIOtask;
28     CPU_TASK_ASSIGN_L(AvailableCPUSlot);
29     IO_TASK_ASSIGN_L(AvailableIOSlot).
```

3.3 Dynamic JASL Algorithm

Although the JASL algorithm can achieve a balance between performance and data locality, it still exhibits some problems, necessitating its modification. Network latency and inappropriate data block examination prolong task execution times, thus affecting the execution of Algorithm 2 or 3, and consequently preventing the *JobTracker* from accurately setting the number of CPU and I/O slots for each *TaskTracker*. Therefore, some *TaskTracker*s with high capability receive a low number of CPU or I/O slots, resulting in the waste of CPU or I/O resources. Conversely, some *TaskTracker*s with low capability receive such a high number of CPU or I/O slots that they have numerous additional tasks, reducing the performance of the Hadoop system. To prevent resource waste, a dynamic adjustment mechanism was added to the JASL algorithm, resulting in Algorithm 10.

Algorithm 10 is an enhanced algorithm called the *dynamic job allocation scheduler with locality* and includes two accumulative parameters, namely *CPUcount* and *IOcount*, that are used to quantify the overload frequency of CPU and I/O utilization. When a *TaskTracker* sends a *Heartbeat* message, the *JobTracker* executes job classification and sets the number of the slots for each *TaskTracker*, in addition to monitoring the CPU utilization and I/O read/write rate of each *TaskTracker*. If the CPU utilization exceeds a certain value (default: 90 percent), the *CPUcount* is increased by one. In addition, if the I/O read/write rate exceeds a certain value (default: 35 MB), the *IOcount* is increased by one. These steps are repeated until the *CPUcount* or *IOcount* value exceeds the threshold, during which the *JobTracker* determines that slots for some *TaskTrackers* are inappropriately set; hence, the *JobTracker* re-executes Algorithm 2 to obtain the accurate slot settings for each *TaskTracker*. This modified procedure enables the *JobTracker* to adjust the CPU or I/O slots for each *TaskTracker* dynamically. As demonstrated in Section 4, the proposed algorithm increases the resource utilization of each *TaskTracker* and improves the overall performance of the Hadoop system.

Algorithm 10. Dynamic_Job_Allocation_Scheduler_with_Locality (DJASL)

```
1  When a batch of jobs are submitted into JobTracker:
2  add jobs into Job Queue;
3  Initialize SetTaskTrackerCPUTable := 0;
4  Initialize SetTaskTrackerIOTable := 0;
5  Initialize CPUcount := 0;
6  Initialize IOcount := 0;
7  Initialize LocalityBenifitTable := 0;
8  while receive Heartbeat by TaskTracker do
9      TaskTrackerCPUslot := 0;
10     TaskTrackerIOslot := 0;
11     obtain TaskTrackerRunningCPUtask from Heartbeat information;
12     obtain TaskTrackerRunningIOtask from Heartbeat information;
13     AvailableCPUSlots := 0;
14     AvailableIOSlots := 0;
15     JOB_CLASSIFICATION_L(Heartbeat);
16     if SetTaskTrackerCPUTable == 1 then
17         obtain TaskTrackerCPUslot according to TaskTrackerCPUTable;
18     else
19         TaskTrackerCPUslot :=SET_CPU_SLOT(Job Queue);
20     if SetTaskTrackerIOTable == 1 then
21         obtain TaskTrackerIOslot according to TaskTrackerIO Table;
22     else
23         TaskTrackerIOslot :=SET_IO_SLOT(Job Queue);
24     if TaskTrackerCPUslot == 0 then
25         TaskTrackerCPUslot := default CPU slot;
26     if TaskTrackerIOslot == 0 then
27         TaskTrackerIOslot := default I/O slot;
28     if TaskTracker.CPUusage $>$ 90% of CPU usage then
29         CPUcount += 1;
30     if TaskTracker.IOusage $>$ 35MB then
31         IOcount += 1;
32     if CPUcount $>=$ 100 then
33         reset CPU_slot;
34     if IOcount $>=$ 100 then
35         reset I/O_slot;
36     AvailableCPUSlots := TaskTrackerCPUslot− TaskTracker RunningCPUtask;
37     AvailableIOSlots := TaskTrackerIOslot− TaskTracker RunningIOtask;
38     CPU_TASK_ASSIGN_L(AvailableCPUSlot);
39     IO_TASK_ASSIGN_L(AvailableIOSlot).
```

TABLE 5
Heterogeneous CPU Experimental Environment

	Master		Slave	
	Quantity	Specification	Quantity	Specification
Environment 1	1	2 cpu & 4 GB memory	33	1 cpu & 2 GB memory
			33	1 cpu & 2 GB memory
			33	1 cpu & 2 GB memory
Environment 2	1	2 cpu & 4 GB memory	33	2 cpu & 2 GB memory
			33	4 cpu & 2 GB memory
			33	8 cpu & 2 GB memory

4 Performance Evaluation

This section presents the experimental environment considered in Section 4.1 and the experimental results regarding the JAS algorithm presented in Section 3.

4.1 Experimental Environment

The experimental setup included an IBM Blade Center H23 with 7 Blades (84 CPUs, 1,450 GB of memory, and 3 TB of disk space divided into four partitions) and a Synology DS214play NAS was mounted to extend hardware resources (Intel Atom CE5335 CPU, and 3 TB of disk space). Moreover, VirtualBox 4.2.8 was used to create several virtual machines. One of the machines served as the master machine and the remaining ones served as slaves. Several heterogenous experimental environment setups (Tables 5, 6 and 7) were employed to observe performance diversification and data locality. We setup different heterogeneous experimental environment each with 100 VMs separated to four partitions of disk space. IBM Blade Center with seven Blades share these partitions together. In order to create diversity of data locality problem, slave nodes are average setup on each disk partitions. The first setup, Environment 1, comprised one master machine, which contained two CPUs with 4 GB of memory, and 99 slave machines, with one CPU each, with 2 GB of memory. Environment 1 was set to the control group. The second setup, Environment 2,

TABLE 6
Heterogeneous Ram Experimental Environment

	Master		Slave	
	Quantity	Specification	Quantity	Specification
Environment 1	1	2 cpu & 4 GB memory	33	1 cpu & 2 GB memory
			33	1 cpu & 2 GB memory
			33	1 cpu & 2 GB memory
Environment 3	1	2 cpu & 4 GB memory	33	1 cpu & 2 GB memory
			33	1 cpu & 4 GB memory
			33	1 cpu & 8 GB memory

TABLE 7
Heterogeneous Master Experimental Environment

	Master		Slave	
	Quantity	Specification	Quantity	Specification
Environment 1	1	2 cpu & 4 GB memory	33	1 cpu & 2 GB memory
			33	1 cpu & 2 GB memory
			33	1 cpu & 2 GB memory
Environment 4	1	4 cpu & 8 GB memory	33	1 cpu & 2 GB memory
			33	1 cpu & 2 GB memory
			33	1 cpu & 2 GB memory

comprised one master, which contained two CPUs with 4 GB of memory, and 99 slaves. Among these 99 slaves, 33 comprised two CPUs with 2 GB of memory, 33 had four CPUs with 2 GB of memory, and 33 had eight CPUs with 2 GB of memory. Environment 2 was established to assess the effect of distinct numbers of CPUs on various group of slaves. The third setup, Environment 3, comprised one master, containing two CPUs with 4 GB of memory, and 99 slaves. Among these 99 slaves, 33 had one CPU with 2 GB of memory, 33 had one CPU with 4 GB of memory, and 33 had one CPU with 8 GB of memory. Environment 3 was established to evaluate the effect of various memory capacities on different groups of slaves. The final setup, Environment 4, comprised one master containing four CPUs with 8 GB of memory and 99 slaves with one CPU and 2 GB of memory. Environment 4 was established to assess the various computing ability of the master node. All machines shared 6 TB of hard disk space. Ubuntu 14.04 LTS was adopted as the operating system. Hadoop-0.20.205.0 was used, and each node comprised four map slots and one reduce slot.

Eight job types were executed: Pi, Wordcount, Terasort, Grep, Inverted-index, Radixsort, Self-join, and K-Means. The data size of different jobs was designed from 5 to 100 GB. When a client sent a request to Hadoop to execute these eight jobs, they were executed in a random order. Ten requests were sent to determine the average job execution time. Section 4.2 reports the results.

4.2 Results

The experimental results can be classified into three themes presented in three sections: 1) Section 4.2.1 presents the individual performance of each job and indicates the effect of various data sizes; (2) Section 4.2.2 shows that the JAS algorithm improves the overall performance of the Hadoop system and that the JASL algorithm improves the data locality of the JAS; and 3) Section 4.2.3 indicates that the proposed DJASL algorithm improves the overall performance of the Hadoop system and that this algorithm has similar data locality to the JASL algorithm.

4.2.1 *Individual Performance of Each Workloads*

Fig. 6 illustrates the individual performance of each jobs, and each job setup comprised nearly 10 GB of data. The

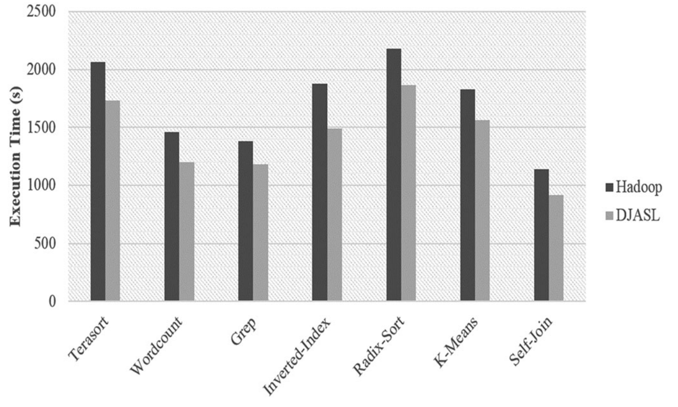


Fig. 6. Average execution time of each job in the Hadoop system and DJASL algorithm.

average execution time of the DJASL was compared with that of the default Hadoop algorithm in Environment 1 (Table 5); the results revealed that the sorting type jobs registered a higher execution time than the other jobs did, and that the join type jobs exhibited a shorter execution time. However, as shown in Fig. 7, when multiple data were batch processed, the execution time did not increase multiples in continuation of the experiment. For example, if we have double data size of workloads, but the execution time will increase less than two times. We allocated nearly 100 GB of data storage space for each request involving different jobs and processed them in batches. The following sections present the experimental results.

4.2.2 *Performance and Data Locality of the JAS and JASL Algorithms*

In some of the ten requests, the performance of the JAS algorithm was not superior to those of Hadoop and DMR because the JAS algorithm sets slots inappropriately. Therefore, the resource utilizations of some *TaskTracker*s became overloaded, and some tasks could not be executed until resources were released. Hence, the execution times of these tasks increased, causing the performance of the JAS algorithm to decrease compared with those of Hadoop and DMR. However, to simulate real situations, the average execution times of all jobs over ten requests were derived. In the heterogeneous computing environment, average execution times of the JAS and JASL algorithms were shorter than those of Hadoop and DMR.

Fig. 8a depicts the average execution times of Hadoop, DMR, and the JAS and JASL algorithms. Because a substantial difference was observed in the CPU and memory

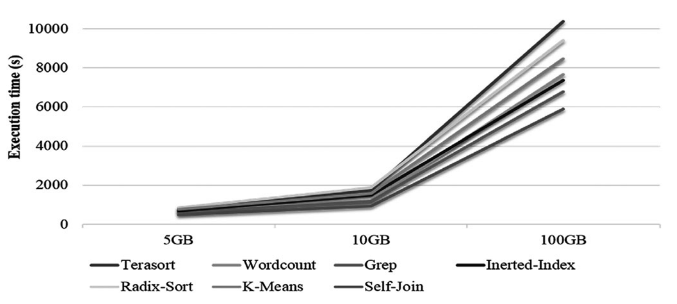
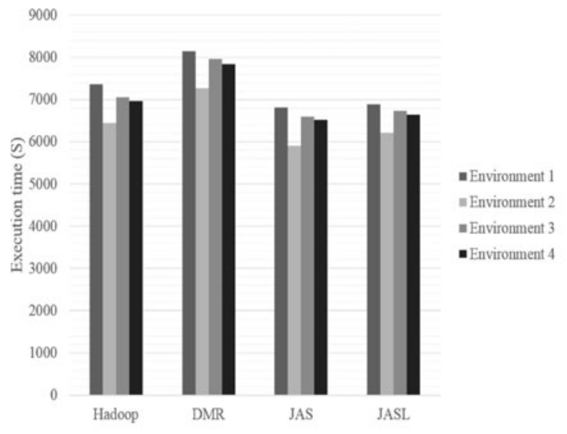
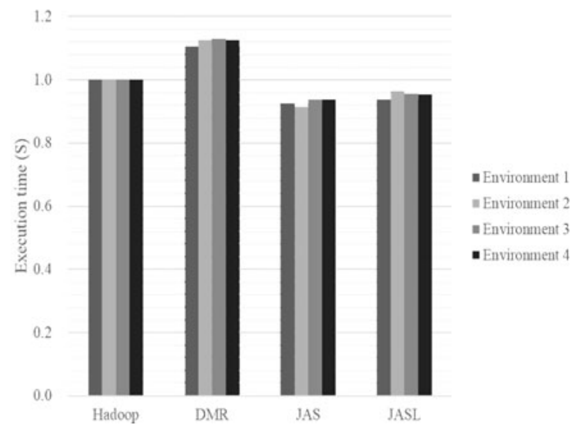


Fig. 7. Average execution time of each job for various data sizes.



(a) Average execution time of a request compared with Hadoop and DMR.



(b) Percentage average execution time of a request compared with Hadoop and DMR.

Fig. 8. Performance of JAS and JASL compared with Hadoop and DMR in four computing environments.

resources between the nodes in those Environments (Tables 5, 6 and 7), the performance of the algorithms in Environment 2 was superior to that of the algorithms in the other environments. However, the difference in performance between the environments was small. The execution time of the JASL algorithm was longer than that of the JAS

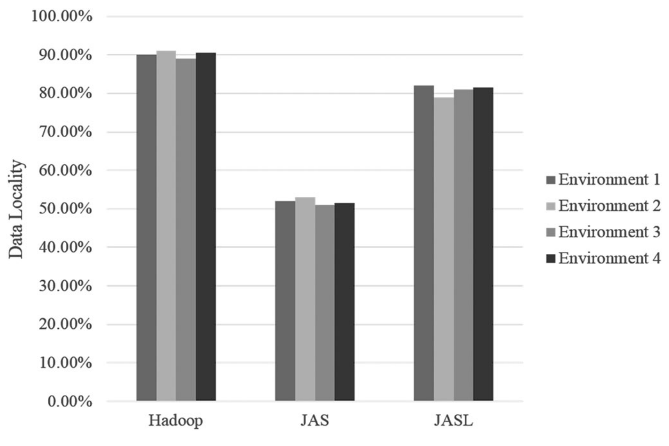


Fig. 9. Data locality of JAS and JASL compared with Hadoop in four environments.

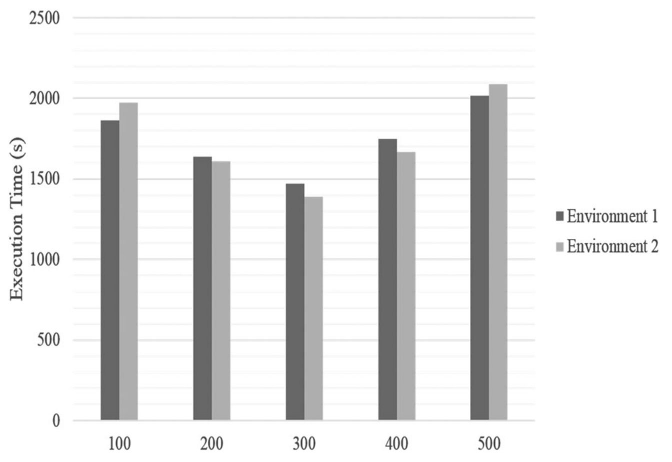


Fig. 10. Average execution time of a job in DJASL for setting the various thresholds.

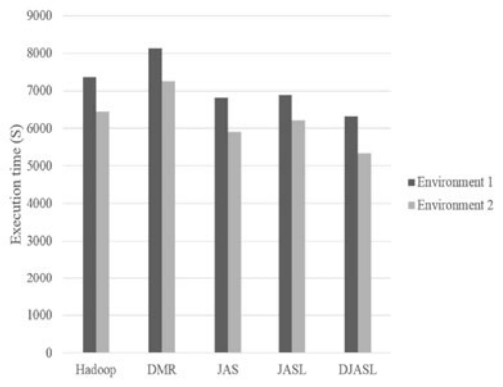
algorithm, but the data locality of the JASL algorithm was substantially greater than that of the JAS algorithm (Fig. 9). Thus, the large amount of extraneous network transformation produced by the JAS algorithm can be reduced. Because of the large processing capability difference between the nodes in Environment 2, higher numbers of efficient nodes were assigned for higher numbers of tasks, reducing data locality.

Fig. 8b illustrates the percentage execution time relative to Hadoop. In the four environments, the performance of the JAS algorithm improved by nearly 15-18 percent compared with Hadoop and nearly 18-20 percent compared with DMR. Moreover, the data locality of the JASL algorithm improved by nearly 25-30 percent compared with the JAS in these environments.

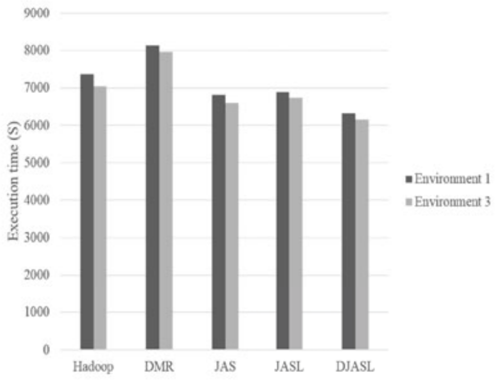
4.2.3 *Performance and Data Locality of the DJASL Algorithm*

The *JobTracker* occasionally inaccurately sets the slots when the JASL algorithm is applied, potentially reducing the performance. Hence, the DJASL algorithm includes two parameters, namely *CPUcount* and *IOcount*, which are used to ensure accurate slot settings. The *JobTracker* resets slots according to threshold values, and differences in the threshold values cause performance results to vary. If a threshold value is too high (i.e., slots are set incorrectly when the DJASL is applied), the *JobTracker* must wait for a long period to reset the slots. By contrast, if a threshold value is too low, the *JobTracker* must reset slots frequently. Inappropriate threshold settings hinder the maximization of resource utilization and negatively affect the performance of the Hadoop system. Therefore, an experiment was conducted in this study to determine the values of various threshold settings. The threshold was set to 100, 200, 300, 400, and 500. According to these five values, five requests were sent to Hadoop, and each request contained ten disordered jobs (five Wordcount and five Terasort). According to Fig. 10, setting the threshold value to 300 yielded the optimal performance.

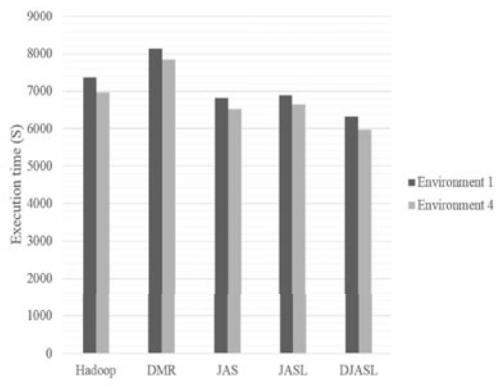
Because the DJASL algorithm can reset slots through a count mechanism, its performance was superior to that of Hadoop. On average, the performance of the DJASL algorithm was superior to that of DMR. However, the performance of the DJASL algorithm was occasionally inferior to



(a) Average execution time of the DJASL compared with the JAS and JASL in heterogeneous environments composed of different numbers of CPUs.



(b) Average execution time of the DJASL compared with the JAS and JASL in heterogeneous environments composed of different amounts of memory.



(c) Percentage of average execution time of DJASL compared with the JAS and JASL (one type of jobs).

Fig. 11. Performance of the DJASL compared with the JAS and JASL in different heterogeneous computing environments.

that of DMR because slots must be reset. In some scenarios, tasks executed by *TaskTracker*s are not removed by the *JobTracker*. Therefore, the *JobTracker* must wait for such tasks to be completed. When *TaskTracker*s become overloaded, the contained tasks cannot be completed until resources are

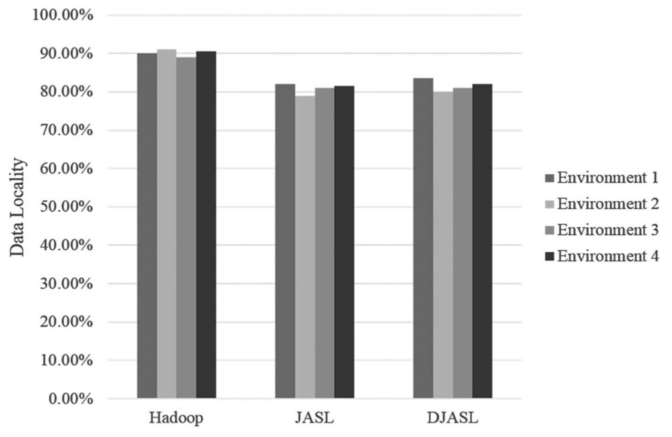


Fig. 12. Data locality of the DJAS compared with JASL and Hadoop in four different environments.

released. Therefore, the execution times for these tasks are prolonged, reducing the performance of the DJASL algorithm compared with that of DMR. When the slots of the *JobTracker* have been reset, the resources of each *TaskTracker* can be used to improve the performance of the Hadoop system. The average execution time of all jobs was used to simulate real situations.

We implemented three heterogeneous computing environments (Tables 5, 6 and 7) and compared each of them in detail with all the presented algorithms (e.g., Hadoop, DMR, JAS, JASL, DJASL). Fig. 11a shows a comparison of the performance of the DJASL algorithm in Environment 2, which comprised a higher number of CPUs in slave computers compared with the master computers, and Environment 1. Fig. 11b depicts a comparison of the performance of the DJASL algorithm in Environment 3 in which more memory was allocated to the slave computers compared with the master computers, and Environment 1. Fig. 11c depicts a comparison of the performance of the DJASL algorithm in Environment 4, in which a higher number of CPUs and memory was allocated to the master node compared with the slave node, and Environment 1. A comparison of the results in Fig. 11 revealed that the numbers of CPUs demonstrated a considerably greater effect on performance regarding the amount of memory resources and improved processing capability of the master node. As shown in Fig. 11, the performance of the DJASL algorithm improved by approximately 27-32 percent compared with DMR and by approximately 16-21 percent compared with Hadoop.

The four heterogeneous computing environments were compared, and Fig. 12 illustrates the results. The data locality of the DJASL algorithm was nearly identical to that of the JASL algorithm. In these environments, the JASL and DJASL effectively improved the data locality and also reduced the differences between these algorithms and Hadoop.

5 Conclusion

This paper proposes job scheduling algorithms to provide highly efficient job schedulers for the Hadoop system. Job types are not evaluated in the default job scheduling policy of Hadoop, causing some *TaskTracker*s to become overloaded. According to the proposed DJASL algorithm, the

JobTracker first computes the capability of each *TaskTracker* and then sets the numbers of CPU and I/O slots accordingly. In addition, the DJASL algorithm substantially improves the data locality of the JAS algorithm and resource utilization of each *TaskTracker*, improving the performance of the Hadoop system. The experimental results revealed that performance of the DJASL algorithm improved by approximately 18 percent compared with Hadoop and by approximately 28 percent compared with DMR. The DJASL also improved the data locality of the JAS by approximately 27 percent.

The proposed scheduling algorithms for heterogeneous cloud computing environments are independent of systems supporting the MapReduce programming model. Therefore, they are not only useful for Hadoop as demonstrated in this paper, but also applicable to other cloud software systems such as YARN and Aneka.

Acknowledgments

This research was partly supported financially by the Headquarters of University Advancement at National Cheng Kung University, which is sponsored by the Ministry of Education, Taiwan, R.O.C. S.-Y. Hsieh is the correspondence author.

References

[1] (2016). Apache Hadoop. [Online]. Available: http://hadoop.apache.org/

[2] (2014). Apache Hadoop YARN. [Online]. Available: http://hadoop.apache.org/docs/current/hadoop-yarn/hadoop-yarn-site/YARN.html

[3] (2016). Hadoop's Capacity Scheduler. [Online]. Available: https://hadoop.apache.org/docs/r2.7.2/hadoop-yarn/hadoop-yarn-site/CapacityScheduler.html

[4] M. Zaharia. (2015). "The hadoop fair scheduler" [Online]. Available: https://hadoop.apache.org/docs/r2.7.1/hadoop-yarn/hadoop-yarn-site/FairScheduler.html

[5] F. Ahmad, S. T. Chakradhar, A. Raghunathan, and T. N. Vijaykumar, "Tarazu: Optimizing mapreduce on heterogeneous clusters," *ACM SIGARCH Comput. Archit. News*, vol. 40, no. 1, pp. 61–74, 2012.

[6] M. J. Atallah, C. Lock, D. C. Marinescu, H. J. Siegel, and T. L. Casavant, "Co-scheduling compute-intensive tasks on a network of workstations: Model and algorithms," in *Proc. 11th Int. Conf. Distrib. Comput. Syst.*, 1991, pp. 344–352.

[7] A. Bezerra, P. Hernandez, A. Espinosa, J. C. Moure, "Job scheduling in Hadoop with shared input policy and RAMDISK," in *Proc. IEEE Int. Conf. Cluster Comput.*, 2014, pp. 355–363.

[8] D. G. Feitelson, and L. Rudolph, "Gang scheduling performance benefitsfor fine-grained synchronization," *J. Parallel Distrib. Comput.*, vol. 16, no. 4, pp. 306–318, 1992.

[9] S. Ghemawat, H. Gobioff, and S. T. Leung, "The google file system," *ACM SIGOPS Operating Syst. Rev.*, vol. 37, no. 5, pp. 29–43, 2003.

[10] D. Ghoshal and L. Ramakrishnan, "Provisioning, placement and pipelining strategies for data-intensive applications in cloud environments," in *Proc. IEEE Int. Conf. Cloud Eng.*, 2014, pp. 325–330.

[11] A. Ghodsi, M. Zaharia, S. Shenker, and I. Stoica, "Choosy: Max-min fair sharing for datacenter jobs with constraints," in *Proc. 8th ACM European Conf. Comput. Syst.*, 2013, pp. 365–378.

[12] M. Hammoud and M. F. Sakr, "Locality-aware reduce task scheduling for MapReduce," in *Proc. IEEE 3rd Int. Conf. Cloud Comput. Tech. Sci.*, 2011, pp. 570–576.

[13] S. Ibrahim, H. Jin, L. Lu, S. Wu, B. He, and Qi, L, "Leen: Locality/fairness-aware key partitioning for mapreduce in the cloud," in *Proc. IEEE 2nd Int. Conf. Cloud Comput Tech. Sci.*, 2010, pp. 17–24.

[14] M. Isard, V. Prabhakaran, J. Currey, U. Wieder, K. Talwar, and A. Goldberg, "Quincy: Fair scheduling for distributed computing clusters," in *Proc. ACM SIGOPS 22nd Symp. Operating Syst. Principles*, 2009, pp. 261–276.

[15] J. K. Ousterhout, "Scheduling techniques for concurrent systems," in *Proc. 3rd Int. Conf. Distrib. Comput. Sys.*, 1982, pp. 22–30.

[16] W. Lee, M. Frank, V. Lee, K. Mackenzie, and L. Rudolph, "Implications of I/O for gang scheduled workloads," in *Proc. Job Scheduling Strategies Parallel Process.*, 1997, pp. 215–237.

[17] H. Lee, D. Lee, and R. S. Ramakrishna, "An enhanced grid scheduling with job priority and equitable interval job distribution," in *Proc. 1st Int. Conf. Grid Pervasive Comput., Lecture Notes Comput. Sci.*, 2006, pp. 53–62.

[18] A. J. Page and T. J. Naughton, "Dynamic task scheduling using genetic algorithms for heterogeneous distributed computing," in *Proc. 19th IEEE Int. Parallel Distrib. Process. Symp.*, 2005, p. 189a.

[19] C. Reiss, A. Tumanov, G. R. Ganger, R. H. Katz, and M. A. Kozuch, "Heterogeneity and dynamicity of clouds at scale: Google trace analysis," in *Proc. 3rd ACM Symp. Cloud Comput.*, 2012, p. 7.

[20] E. Rosti, G. Serazzi, E. Smirni, and M. S. Squillante, "The impact of I/O on program behavior and parallel scheduling," *ACM SIGMETRICS Perform. Eval. Rev.*, vol. 26, no. 1, pp. 56–65, 1998.

[21] E. Rosti, G. Serazzi, E. Smirni, and M. S. Squillante, "Models of parallel applications with large computation and I/O requirements," *IEEE Trans. Softw. Eng.*, vol. 28, no. 3, pp. 286–307, Mar. 2002.

[22] M. Schwarzkopf, A. Konwinski, M. Abd-El-Malek, and J. Wilkes, "Omega: Flexible, scalable schedulers for large compute clusters," in *Proc. 8th ACM European Conf. Comput. Syst.*, 2013, pp. 351–364.

[23] C. Tian, H. Zhou, Y. He, and L. Zha, "A dynamic mapreduce scheduler for heterogeneous workloads," in *Proc. 8th IEEE Int. Conf. Grid Cooperative Comput.*, 2009, pp. 218–224.

[24] A. Tumanov, J. Cipar, G. R. Ganger, and M. A. Kozuch, "Alsched: Algebraic scheduling of mixed workloads in heterogeneous clouds," in *Proc. 3rd ACM Symp. Cloud Comput.*, 2012, p. 25.

[25] J. Weinman, "Cloud computing is NP-complete" in *Proc. Tech. Symp. ITU Telecom World*, 2011, pp. 75–81.

[26] Y. Wiseman, and D. G. Feitelson, "Paired gang scheduling," *IEEE Trans. Parallel Distrib. Syst.*, vol. 14, no. 6, pp. 581–592, Jun. 2003.

[27] M. Zaharia, D. Borthakur, S. J. Sarma, K. Elmeleegy, S. Shenker, and I. Stoica, "Delay scheduling: A simple technique for achieving locality and fairness in cluster scheduling," in *Proc. 5th Eur. Conf. Comput. Syst.*, 2010, pp. 265–278.

[28] X. Zhang, Z. Zhong, S. Feng, B. Tu, and J. Fan, "Improving data locality of mapreduce by scheduling in homogeneous computing environments," in *Proc. 9th IEEE Int. Symp. Parallel Distrib. Process. Appl.*, 2011, pp. 120–126.

Sun-Yuan Hsieh received the PhD degree in computer science from National Taiwan University, Taipei, Taiwan, in June 1998. He then served the compulsory two-year military service. From August 2000 to January 2002, he was an assistant professor at the Department of Computer Science and Information Engineering, National Chi Nan University. In February 2002, he joined the Department of Computer Science and Information Engineering, National Cheng Kung University, and now he is a distinguished professor. He received the 2007 K. T. Lee Research Award, President's Citation Award (American Biographical Institute) in 2007, the Engineering Professor Award of Chinese Institute of Engineers (Kaohsiung Branch) in 2008, the National Science Council's Outstanding Research Award in 2009, and IEEE Outstanding Technical Achievement Award (IEEE Tainan Section) in 2011. He is a fellow of the British Computer Society (BCS). His current research interests include design and analysis of algorithms, fault-tolerant computing, bioinformatics, parallel and distributed computing, and algorithmic graph theory. He is a senior member of the IEEE

Chi-Ting Chen received the MS degree in the Department of Computer Science and Information Engineering from National Cheng Kung University, Tainan, in 2015. His research interests include hub location problem, cloud computing and design and analysis of algorithms.

Chi-Hao Chen received the MS degree in the Department of Computer Science and Information Engineering from National Cheng Kung University, Tainan, in 2014. His research interests include cloud computing and design and analysis of algorithms.

Tzu-Hsiang Yen received the MS degree in the Department of Computer Science and Information Engineering from National Cheng Kung University, Tainan, in 2013. His research interests include cloud computing and design and analysis of algorithms.

Hung-Chang Hsiao received the PhD degree in computer science from National Tsing-Hua University, Taiwan, in 2000. He is a full professor in Computer Science and Information Engineering, National Cheng-Kung University, Taiwan, since August 2012. His research interests include distributed and parallel computing, NoSQL databases, big data, and randomized algorithm design and performance analysis.

Dr. Rajkumar Buyya is a professor of Computer Science and Software Engineering, future fellow of the Australian Research Council, and the Director of the Cloud Computing and Distributed Systems (CLOUDS) Laboratory at the University of Melbourne, Australia. He is also serving as the founding CEO of Manjrasoft, a spin-off company of the University, commercializing its innovations in Cloud Computing. He has authored more than 500 publications and six text books including "Mastering Cloud Computing" published by McGraw Hill, China Machine Press, Morgan Kaufmann, and for Indian, Chinese, and international markets respectively. He is one of the highly cited authors in computer science and software engineering worldwide. Software technologies for Grid and Cloud computing developed under his leadership have gained rapid acceptance and are in use at several academic institutions and commercial enterprises in 40 countries around the world. He has led the establishment and development of key community activities, including serving as foundation Chair of the IEEE Technical Committee on Scalable Computing and five IEEE/ACM conferences. These contributions and international research leadership of him are recognized through the award of "2009 IEEE TCSC Medal for Excellence in Scalable Computing" from the IEEE Computer Society TCSC. Manjrasoft's Aneka Cloud technology developed under his leadership has received "2010 Frost and Sullivan New Product Innovation Award" and recently Manjrasoft has been recognised as one of the Top 20 Cloud Computing companies by the Silicon Review Magazine. He is currently serving as Co-Editor-in-Chief of Journal of Software: Practice and Experience, which was established 40+ years ago. He is a fellow of the IEEE.