International Journal of Scientific Engineering and Research (IJSER)
ISSN (Online): 2347-3878 (UGC Approved, Sr. No. 48096)
Index Copernicus Value (2015): 62.86 | Impact Factor (2015): 3.791

Deploying Efficiently MapReduce Applications in Heterogeneous Computing Environments using Novel Scheduling Algorithms

Dr. Shubhangi D C[1], Bushra Sultana [2]

[1]Head of the Department, Department of Computer Science & Engineering, VTU PG Centre, Kalaburgi, Karnataka, India

[2]P.G.Student, Department of Computer Science & Engineering, VTU PG Centre, Kalaburgi, Karnataka, India

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 Map Reduce 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 CPUbound 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*

Keywords: Hadoop, Heterogeneous environments, Heterogeneous workloads, Map Reduce and 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 Map Reduce framework designed for parallelizing large data sets and splitting them into thousands of processing nodes in a cluster is a crucial concept. Hadoop which implements the MapReduce programming framework, is an opensource 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 Job Tracker, manages multiple slave nodes, called ask Trackers, to process tasks assigned by the Job Tracker1.

Modules

- Individual Performance of Each Workload.
- Performance analysis of the Job Allocation Scheduler and Job Allocation Scheduler Locality.
- Performance of the Dynamic Job Allocation and Scheduler Locality.

Individual Performance of Each Workload

The individual performance of each jobs, and each job setup comprised nearly 10 GB of data. The average execution time of the DJASL was compared with that of the default Hadoop algorithm in Environment 1 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 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.

Performance analysis of the Job Allocation Scheduler and Job Allocation Scheduler Locality

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 Task Trackers 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.

Performance of the Dynamic Job Allocation and Scheduler Locality

The Job Tracker occasionally inaccurately sets the slots when the JASL algorithm is applied, potentially reducing the performance. Hence, the DJASL algorithm includes two parameters, namely CPU count and IO count, which are used to ensure accurate slot settings. The Job Tracker 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 Job Tracker must wait for a long

period to reset the slots. By contrast, if a threshold value is too low, the Job Tracker must reset slots frequently2.

2. System Design

Hadoop Default Scheduler Hadoop supports the Map Reduce 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). Map Reduce 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 Map Reduce job to a Job Tracker, the Job Tracker adds the job to the Job Queue.

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

Step 3
Task assignment
When a Task Tracker periodically (every 3 seconds by default) sends a Heartbeat to a Job Tracker, the Job Tracker obtains information on the current state of the Task Tracker to determine whether it has available slots.

Job Workloads
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.

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 Task Trackers. Executing some of these tasks may require only CPU or I/O resources.

Because the default job scheduler in Hadoop does not balance resource utilization, some tasks in the Task Tracker 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.

Dynamic Map-Reduce Scheduler
To address the imbalanced resource allocation problem of the default scheduler in Hadoop, as described in Section 2.3, 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 Job Trackers by using CPU- and I/Obound queues.

It then assigns two CPU-bound job tasks, J1t1 and J1t2, and two I/O-bound job tasks, J2t1 and J2t2, to Task Tracker1.

Task Tracker3 can execute one CPU-bound job and three I/O-bound jobs simultaneously (i.e., Task Tracker3 has three CPU slots and one I/O slot). Nevertheless, according to the DMR approach, each Task Tracker has two CPU slots and two I/O slots (implying a total of four slots). After receiving jobs from clients, the Job Tracker assigns the tasks to a Task Tracker. Each Task Tracker contains two CPUbound tasks and two I/O-bound tasks. Therefore, Task Tracker1 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. Task Tracker2
has one CPU slot that must wait for CPU resources to be released; therefore, the CPU capacity of Task Tracker2 becomes overloaded. Finally, Task Tracker3 has one CPU slot that must wait for CPU resources to be released; therefore, the CPU capacity of Task Tracker3 becomes overloaded. Furthermore, Task Tracker3 includes one idle I/O slot, indicating 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.

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.

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

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

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 := 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.
```

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.

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 _ 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.

Table 2: Parameters Used for Setting CPU Slots

Notation	Meaning
$T = \{t_1, t_2, ..., t_{n-1}, t_n\}, \lvert T\rvert = n$	the set of tasks on a CPU-bound job
ET_{t_i}	the execution time of task t_i, where $t_i \in T$
$M_i = \{t_j \in T \mid t_j \text{ runs on } TaskTracker_i\}$	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 $TaskTracker_i$
$r_j, j = 1, ..., m$	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 $TaskTracker_y$

Algorithm 2: SET_CPU_SLOT(*Job Queue*)

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

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

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

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 TaskTracker CPUCapability. 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 TaskTracker CPUCapability 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.

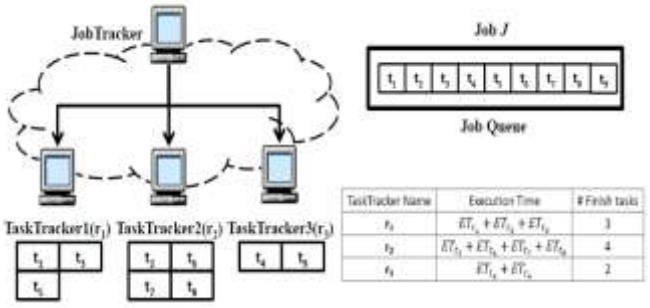


Figure 4: Illustration of Algorithm 2

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 TaskTrackers: r1, r2, and r3. The JobTracker distributes the nine tasks such that t1, t3, and t5 are assigned to r1; t2, t6, t7, and t8 are assigned to r2; and t4 and t9 are assigned to r3. 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 t1 has been completed and then the JobTracker classifies it as a CPU-bound job, the JobTracker records its execution time in r1 in TaskTrackerCPUCapability, therefore, Task- TrackerCPUCapability contains the record that t1 has been completed by r1. These steps are repeated for recording t2–t9 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 r1 is c1 = t1+t3+t5 3 ; the capacity of r2 is c2 = t2+t6+t7+t8 4 ; and the capacity of r3 is c3 = t4+t9 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 r1 is k1 = s 2 _ c1 c1+c2+c3 ; the number of CPU slots in r2 is k2 = s 2 _ c2 c1+c2+c3 ; and the number of CPU slots in r3 is k3 = s 2_ c3c1+c2+c3 . 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 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.

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.

Table 3: Parameters Used for Setting I/O Slots

Notation	Meaning
$L = \{t_1, t_2, ..., t_n\}, \lvert L\rvert = n$	the set of tasks on a I/O-bound job
l_{t_i}	the execution time of task t_i, where $t_i \in L$
$N_i = \{t_j \in L \mid t_j \text{ run on } TaskTracker_i\}$	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 $TaskTracker_y$

Suppose that m TaskTrackers exist: TaskTracker1; TaskTracker2; : : : ; TaskTrackerm. Define

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

```
for Job in Job Queue do
    if Job has been finished and Job is I/O-bound then
        obtain the task information from TaskTracker;
        compute the TaskTracker capability according to TaskTrackerIOCapability;
        d_y := n_y / f_y;
        for each TaskTracker do
            i_y := (the number of I/O slots) * d_y / (sum_{j=1}^{m} d_j);
            record i_y on TaskTrackerIOTable;
            SetTaskTrackerIOTable := 1;
        return TaskTrackerIOslot according to TaskTrackerIOTable;
        break.
```

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

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 TaskTrackerI-OCapability 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.

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.

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

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;
```

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 the JobTracker to allocate tasks accurately to the idle CPU slots of TaskTrackers, 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. 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

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 TaskTrackerQueues of a TaskTracker according to the Heartbeat information;
5              break;
6      if ∃ a task is not selected and moved to the TaskTrackerQueues 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 TaskTrackerQueues 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 TaskTrackerQueues of the corresponding TaskTracker according to the information of
                       Heartbeat;
15                     break;
```

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 TaskTrackers, thus preventing the waste of idle I/O slots.

3.1.6. Complete JAS Algorithm

All the algorithms presented in the preceding subsections 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/Obound queue are assigned to TaskTrackers. 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 Tasktrackers. Because of the property of the JAS, data locality is lost and a substantial amount of network traffic occurs. Fig. 5

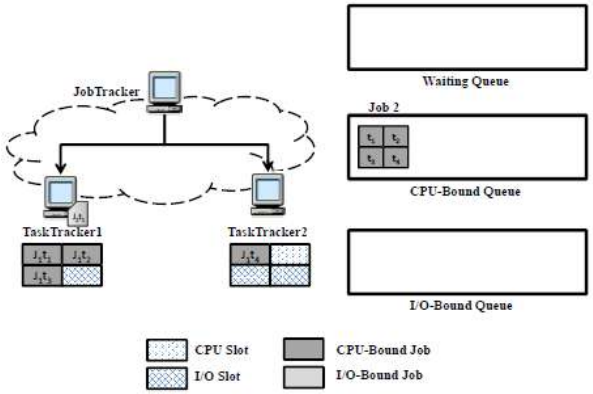
Algorithm 6: Job_Allocation_Scheduler (JAS)

```
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 − TaskTrackerRunningCPUtask;
26     AvailableIOSlots:=TaskTrackerIOslot − TaskTrackerRunningIOtask;
27     CPU_TASK_ASSIGN(AvailableCPUSlot);
28     IO_TASK_ASSIGN(AvailableIOSlot).
```

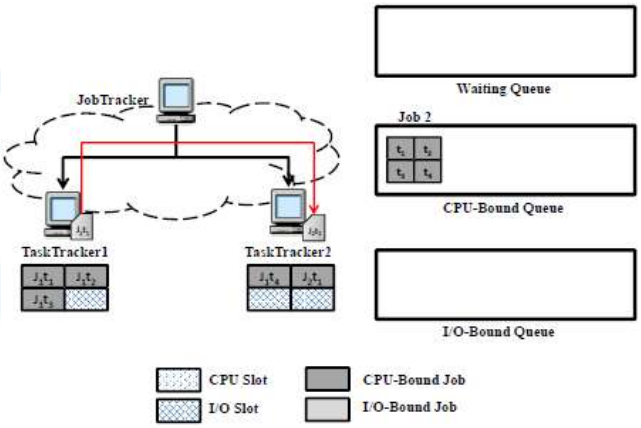
Table 4: JAS Parameters

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

illustrates these problems.



(a) JobTracker assign tasks of Job 1 according to Task-Trackers 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 J2t1 to TaskTracker2. Because the data block required by J2t1 is in JobTracker1, it needs to be moved from TaskTracker1 to TaskTracker2, which may increase network traffic.

Figure 5: JAS algorithm may increase network traffic.

Assume that TaskTrackers are available (i.e., TaskTracker1 and TaskTracker2); TaskTracker1 has three CPU slots and one I/O slot, and TaskTracker2 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 J1t1, J1t2, and J1t3 to TaskTracker1, and assigns J1t4 to TaskTracker2 (Fig. 5(a)). Job 2 remains in the CPU-bound queue; subsequently, the JobTracker assigns its tasks. When the JAS is applied, the JobTracker tends to assign J2t1 to TaskTracker2. However, because the data block required by J2t1 is in TaskTracker1; TaskTracker2 must retrieve this block from TaskTracker1. The transfer of the data block results in extra network traffic, eliminating the benefit of data locality (Fig. 5(b)).

3.2.2. Improving the JAS

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 = 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;
```

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 Task- Trackers in LocalityBene_tTable 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 NonLocalBene_t is true, then the TaskTracker can execute a non-local task; otherwise, it can execute only local tasks, as indicated by Algorithm 8.

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

Let ETi and ETj be the execution times of two TaskTrackers in a cluster, and let Transfer time be the transfer time of a block from TaskTrackeri to TaskTrackerj . 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 8: CPU_TASK_ASSIGN_L

```
1  for each AvailableCPUSlot do
2      ET1: the execution time of the task executed by the current TaskTracker;
3      ET2: the execution time of the task executed by the remote TaskTracker;
4      Transfer_Time: the transfer time between the TaskTrackers;
5      obtain ET1 and ET2 + 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 TaskTracerQueues 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)

```
When a batch of jobs is submitted to JobTracker:
add jobs into Job Queue;
SetTaskTrackerCPUTable:= 0;
SetTaskTrackerIOTable:= 0;
Initialize LocalityBenifitTable:= 0;
while receive Heartbeat by TaskTracker do
    TaskTrackerCPUslot:=0;
    TaskTrackerIOslot:=0;
    obtain TaskTrackerRunningCPUtask from Heartbeat information;
    obtain TaskTrackerRunningIOtask from Heartbeat information;
    AvailableCPUSlots:= 0;
    AvailableIOSlots:= 0;
    JOB_CLASSIFICATION_L(Heartbeat);
    if SetTaskTrackerCPUTable == 1 then
        obtain TaskTrackerCPUslot according to TaskTrackerCPUTable;
    else
        TaskTrackerCPUslot:=SET_CPU_SLOT(Job Queue);
    if SetTaskTrackerIOTable == 1 then
        get TaskTrackerIOslot according to TaskTrackerIOTable;
    else
        TaskTrackerIOslot :=SET_IO_SLOT(Job Queue);
    if TaskTrackerCPUslot == 0 then
        TaskTrackerCPUslot := default CPU slot;
    if TaskTrackerIOslot == 0 then
        TaskTrackerIOslot := default I/O slot;
    AvailableCPUSlots := TaskTrackerCPUslot − TaskTrackerRunningCPUtask;
    AvailableIOSlots:= TaskTrackerIOslot − TaskTrackerRunningIOtask;
    CPU_TASK_ASSIGN_L(AvailableCPUSlot);
    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 TaskTrackers 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 TaskTrackers 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: Dynamic_Job_Allocation_Scheduler _with_Locality (DJASL)

```
When a batch of jobs are submitted into JobTracker:
add jobs into Job Queue;
Initialize SetTaskTrackerCPUTable := 0;
Initialize SetTaskTrackerIOTable := 0;
Initialize CPUcount := 0;
Initialize IOcount := 0;
Initialize LocalityBenifitTable := 0;
while receive Heartbeat by TaskTracker do
    TaskTrackerCPUslot := 0;
    TaskTrackerIOslot := 0;
    obtain TaskTrackerRunningCPUtask from Heartbeat information;
    obtain TaskTrackerRunningIOtask from Heartbeat information;
    AvailableCPUSlots := 0;
    AvailableIOSlots := 0;
    JOB_CLASSIFICATION_L(Heartbeat);
    if SetTaskTrackerCPUTable == 1 then
        obtain TaskTrackerCPUslot according to TaskTrackerCPUTable;
    else
        TaskTrackerCPUslot :=SET_CPU_SLOT(Job Queue);
    if SetTaskTrackerIOTable == 1 then
        obtain TaskTrackerIOslot according to TaskTrackerIOTable;
    else
        TaskTrackerIOslot :=SET_IO_SLOT(Job Queue);
    if TaskTrackerCPUslot == 0 then
        TaskTrackerCPUslot := default CPU slot;
    if TaskTrackerIOslot == 0 then
        TaskTrackerIOslot := default I/O slot;
    if TaskTracker.CPUusage > 90% of CPU usage then
        CPUcount += 1;
    if TaskTracker.IOusage > 35MB then
        IOcount += 1;
    if CPUcount >= 100 then
        reset CPU_slot;
    if IOcount >= 100 then
        reset I/O_slot;
    AvailableCPUSlots := TaskTrackerCPUslot − TaskTrackerRunningCPUtask;
    AvailableIOSlots := TaskTrackerIOslot − TaskTrackerRunningIOtask;
    CPU_TASK_ASSIGN_L(AvailableCPUSlot);
    IO_TASK_ASSIGN_L(AvailableIOSlot).
```

3. 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.

Individual Performance of Each Workloads

Illustrates the individual performance of each jobs, and each job setup comprised nearly 10 GB of data. The average execution time of the DJASL was compared with that of the default Hadoop algorithm in Environment 1 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 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.

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 Task Trackers

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. 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 resources between the nodes in those Environments (Tables No. 1-3), 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 algorithm, but the data of the JASL algorithm was substantially greater than that of the JAS algorithm (Figure No.6). 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. Illustrates the percentage execution time relative to Hadoop. In the four environments, the performance of the JAS algorithm improved by nearly 15%-18% compared with Hadoop and nearly 18%-20% compared with DMR. Moreover, the data locality of the JASL algorithm improved by nearly 25%-30% compared with the JAS in these environments.

Performance and Data Locality of the DJASL Algorithm

The Job Tracker occasionally inaccurately sets the slots when the JASL algorithm is applied, potentially reducing the performance. Hence, the DJASL algorithm includes two parameters, namely CPU count and IO count, which are used to ensure accurate slot settings. The Job Tracker 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 Job Tracker must wait for a long period to reset the slots. By contrast, if a threshold value is too low, the Job Tracker 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 Word count and five Tera sort). According to Figure No.7, 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 that of DMR because slots must be reset. In some scenarios, tasks executed by Task Trackers are not removed by the Job Tracker. Therefore, the Job Tracker must wait for such tasks to be completed. When Task Trackers become overloaded, the contained tasks cannot be completed until resources are 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 Job Tracker have been reset, the resources of each Task Tracker 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 No.1-3) and compared each of them in detail with all the presented algorithms (e.g., Hadoop, DMR, JAS, JASL, DJASL). Figure No.8 (a) 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. Figure No.8 (b) 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. Figure No.8 (c) 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 Figure No.8 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 Figure No.8, the performance of the DJASL algorithm improved by approximately 27%-32% compared with DMR and by approximately 16%-21% compared with Hadoop. The four heterogeneous computing environments were compared, and 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.

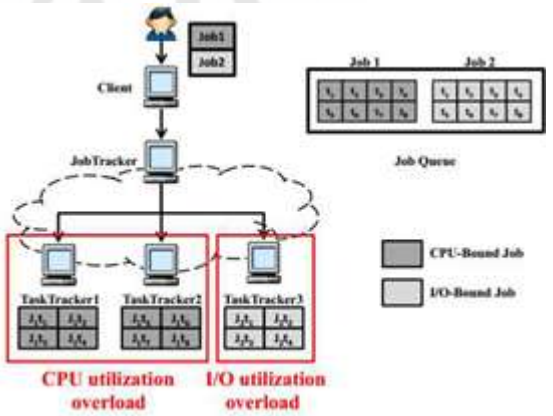
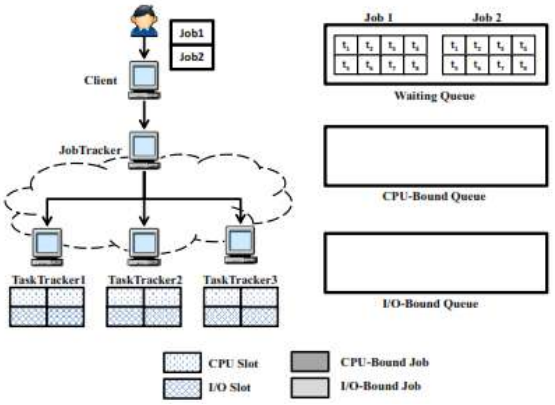
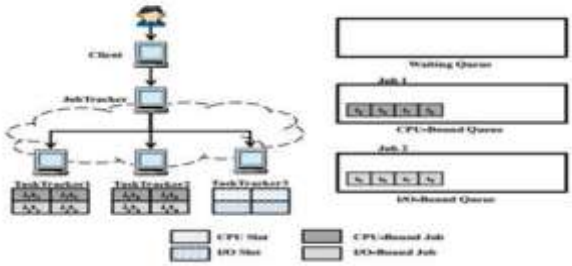


Figure 1: Imbalanced resource allocation

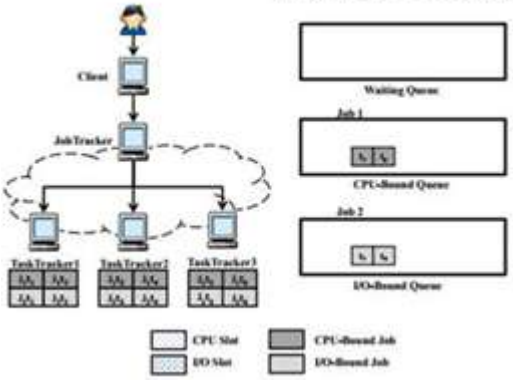


(a) When a client submits a new job, the submitted job is added to the waiting queue. Subsequently, the scheduler classifies the job type

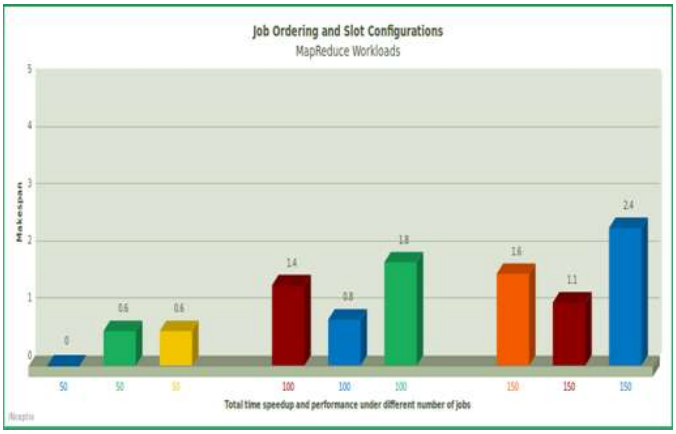


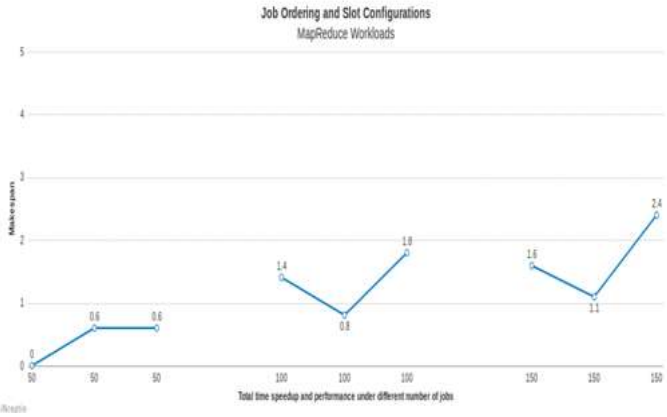
(b) After the scheduler classifies the job type, the jobs are added to the CPU-bound queue or the I/Obound queue. The *Job Tracker* then assigns these tasks ac-cording to the number of free CPU or I/O slots contained in the *Job Tracker*

tained in the *TaskTracker*.



(c) The *Job Tracker* assigns tasks until all of the Task-Trackers have no free slots Figure No.2: Workflow of a DMR scheduler





4. 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 *Task Tracker*s to become overloaded. According to the proposed DJASL algorithm, the *Job Tracker* first computes the capability of each *Task Tracker* 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 *Task Tracker*, improving the performance of the Hadoop system. The experimental results revealed that performance of the DJASL algorithm improved by approximately 18% compared with Hadoop and by approximately 28% compared with DMR. The DJASL also improved the data locality of the JAS by approximately 27%. The proposed scheduling algorithms for heterogeneous cloud computing environments are independent of systems supporting the Map Reduce 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.

References

[1] Apache Hadoop. http://hadoop.apache.org/
[2] Apache Hadoop YARN. http://hadoop.apache.org/docs/current/hadoop-yarn/hadoop-yarn-site/YARN.html
[3] Hadoop's Capacity Scheduler. http://hadoop.apache.org/core/docs/current/capacityscheduler.html.
[4] Matei Zaharia, "The Hadoop Fair Scheduler" http://developer.yahoo.net/blogs/hadoop/FairSharePres.ppt
[5] Ahmad, F., Chakradhar, S. T., Raghunathan, A., and Vijaykumar, T. N., "Tarazu: optimizing mapreduce on heterogeneous clusters," In ACM SIGARCH Computer Architecture News, Vol. 40,No. 1, pp. 61–74, 2012.
[6] Atallah, M. J., Lock, C., Marinescu, D. C., Siegel, H. J., and Casavant, T. L., " Co-scheduling compute-intensive tasks on a network of workstations: model and algorithms," In Proceedings of the 11th International Conference on Distributed Computing Systems, pp. 344–352, 1991.

[7] Bezerra, A., Hernandez, P., Espinosa, A., Moure, J.C., "Job scheduling in Hadoop with Shared Input Policy and RAMDISK," Cluster Computing (CLUSTER), 2014 IEEE International Conference , pp.355–363, 2014.
[8] Feitelson, D. G., and Rudolph, L., "Gang scheduling performance benefitsfor fine-grained synchronization," Journal of Parallel and Distributed Computing, Vol. 16, No.4 , pp. 306–318, 1992.
[9] Ghemawat, S., Gobioff, H., and Leung, S. T., "The Google file system," In ACM SIGOPS Operating Systems Review, Vol. 37, No. 5, pp. 29–43, 2003.
[10] Ghoshal, D., Ramakrishnan, L.,"Provisioning, Placement and Pipelining Strategies for Data-Intensive Applications in Cloud Environments," Cloud Engineering (IC2E), 2014 IEEE International Conference , pp. 325–330, 2014.
[11] Ghodsi, A., Zaharia, M., Shenker, S., and Stoica, I., "Choosy: max-min fair sharing for datacenter jobs with constraints," In Proceedings of the 8th ACM European Conference on Computer Systems, pp. 365–378, 2013.
[12] Hammoud, M., and Sakr, M. F. , "Locality-aware reduce task scheduling for MapReduce," In Proceedings of IEEE Third International Conference on Cloud Computing Technology and Science (Cloud-Com), pp. 570–576, 2011.
[13] Ibrahim, S., Jin, H., Lu, L., Wu, S., He, B., and Qi, L, "Leen: Locality/fairness-aware key partitioning for mapreduce in the cloud," In Proceedings of IEEE Second International Conference on Cloud Computing Technology and Science (CloudCom), pp. 17–24, 2010.
[14] Isard, M., Prabhakaran, V., Currey, J., Wieder, U., Talwar, K., and Goldberg, A., "Quincy: fair scheduling for distributed computing clusters," In Proceedings of the ACM SIGOPS 22nd symposium on Operating systems principles, pp. 261–276, 2009.
[15] J.K. Ousterhout, "Scheduling techniques for concurrent systems," in Proceedings of the third International Conference on Distributed Computing Systems, pp. 22–30, 1982.
[16] Lee, W., Frank, M., Lee, V., Mackenzie, K., and Rudolph, L., "Implications of I/O for Gang Scheduled Workloads," In Proceedings of Springer Berlin Heidelberg on Job Scheduling Strategies for Parallel Processing, pp. 215–237, 1997.
[17] Lee, H., Lee, D., and Ramakrishna, R. S., "An Enhanced Grid Scheduling with Job Priority and Equitable Interval Job Distribution," In Proceedings of the first International Conference on Grid and Pervasive Computing, Lecture Notes in Computer Science, pp. 53–62, 2006.
[18] Page, A. J., and Naughton, T. J., "Dynamic task scheduling using genetic algorithms for heterogeneous distributed computing," In Proceedings of 19th IEEE International on Parallel and Distributed Processing Symposium, pp. 189a–189a, 2005.
[19] Reiss, C., Tumanov, A., Ganger, G. R., Katz, R. H., and Kozuch, M. A., "Heterogeneity and dynamicity of clouds at scale: Google trace analysis," In Proceedings of the Third ACM Symposium on Cloud Computing, pp. 7, 2012.
[20] Rosti, E., Serazzi, G., Smirni, E., and Squillante, M. S., "The Impact of I/O on Program Behavior and Parallel Scheduling," In ACM SIGMETRICS Performance Evaluation Review, Vol. 26, No. 1, pp. 56–65, 1998.