Heterogeneous Job Allocation Scheduler for Hadoop MapReduce Using Dynamic Grouping Integrated Neighboring Search

Chi-Ting Chen, Ling-Ju Hung, Sun-Yuan Hsieh, *Senior Member, IEEE*, Rajkumar Buyya, *Fellow, IEEE*, and Albert Y. Zomaya, *Fellow, IEEE*

Abstract—MapReduce is a crucial framework in the cloud computing architecture, and is implemented by Apache Hadoop and other cloud computing platforms. The resources required for executing jobs in a large data center vary according to the job types. In general, there are two types of jobs, CPU-bound and I/O-bound, which require different resources but run simultaneously in the same cluster. The default job scheduling policy of Hadoop is first-come-first-served and therefore, may cause unbalanced resource utilization. Considering various job workloads, numerous job allocation schedulers were proposed in the literature. However, those schedulers encountered the data locality problem or unreasonable job execution performance. This study proposes a job scheduler based on a dynamic grouping integrated neighboring search strategy, which can balance the resource utilization and improve the performance and data locality in heterogeneous computing environments.

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

1 Introduction

The scale and maturity of the Internet has recently increased dramatically, providing an excellent opportunity for enterprises to conduct business at a global level with minimum investment. Enterprises are rapidly capturing/collecting enormous amount of business data, and therefore, must be able to process data in a timely manner. Scientific and big data applications have similar requirements. Therefore, processing large amount of data in parallel for producing results in timely manner has become more vital. Cloud computing has emerged as a new paradigm for supporting 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 crucial. Hadoop [2], which implements the MapReduce programming framework, is an open-source distributed system that is used by numerous enterprises, including Google, Yahoo, and Facebook for processing large data sets. This study focused on the MapReduce and Hadoop distributed file system (HDFS) layers, which form the core of Apache Hadoop.

HDFS is implemented by Yahoo based on the Google File System, which is used with the MapReduce model. HDFS is a distributed storage system that adopts the master/slave architecture. It comprises a *NameNode* and many *DataNodes*. The *NameNode* is responsible for the management of the entire file system, information of each file such as namespace and metadata, and storage and management. A *DataNode* is responsible for storing data blocks. When an HDFS client reads data from the HDFS, it requests the *NameNode* to determine *DataNodes* that have data blocks that must be read. While writing data, the HDFS client first requests the *NameNode* to create a file. After the *NameNode* accepts this request, the HDFS client directly writes the file to the assigned *DataNodes*.

Using this distributed file system, the read rate is considerably faster than that of a single node. A larger file can be read from many nodes simultaneously, which is more efficient than reading it from a single node read. This not only reduces the time of accessing data, but also reduces the average depletion of a hard disk, which further reduces the cost of hardware maintenance. If the size of the files that must be written to the HDFS is larger than the capacity of one physical hard disk, the files can be divided into several blocks to be stored.

MapReduce mainly uses the divide and conquer policy. It distributes a big data to many nodes for parallel

- *C.-T. Chen and L.-J. Hung are with the Department of Computer Science and Information Engineering, National Cheng Kung University, No. 1, University Road, Tainan 701, Taiwan, R.O.C. E-mail: p78054018@mail.ncku.edu.tw, hunglc@cs.ccu.edu.tw.*
- *S.-Y. Hsieh is with the Department of Computer Science and Information Engineering and Institute of Medical Informatics, National Cheng Kung University, No. 1, University Road, Tainan 701, Taiwan, R.O.C. E-mail: hsiehsy@mail.ncku.edu.tw.*
- *R. Buyya is with the Cloud Computing and Distributed Systems (CLOUDS) Laboratory, School of Computing and Information Systems, University of Melbourne, Parkville VIC 3010, Australia. E-mail: rbuyya@unimelb.edu.au.*
- *A.Y. Zomaya is with the Department of Computer Science and Information Systems, University of Sydney, Camperdown, NSW 2006, Australia. E-mail: albert.zomaya@sydney.edu.au.*

Manuscript received 14 Apr. 2016; revised 9 Feb. 2017; accepted 4 Aug. 2017. Date of publication 4 Sept. 2017; date of current version 11 Mar. 2020. (Corresponding author: Sun-Yuan Hsieh.) Recommended for acceptance by H. Lei. Digital Object Identifier no. 10.1109/TCC.2017.2748586

processing, which reduces the execution time and improves performance. Because of the main distributed scheduler of data blocks processing are in map phase, in order to improving performance by job assignment strategy so we focused on the problem associated with the map phase of MapReduce, which assigns tasks through the default first-come, first-served (FCFS) scheduler.

Hadoop is a server-client architecture system that uses the masterslave concept. The master node, called *JobTracker*, manages multiple slave nodes, called *TaskTracker*s, to process the 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 the input data and output the intermediate data to its local node, whereas reduce tasks receive the intermediate data from several *TaskTracker*s and output the final result.

By default, Hadoop adopts a FCFS job scheduling policy. A *TaskTracker* transfers requests to the *JobTracker* through the *Heartbeat cycle*. Because resource utilization is not considered in the default job scheduling policy of Hadoop (FCFS), *JobTracker* assigns tasks to *TaskTracker* without checking the appropriate resources on *TaskTracker*, some slave nodes do not have the sufficient amount of resources and capacity to perform an assigned task. Therefore, these nodes cannot continue to perform tasks after the system releases resources, leading to poor performance of a Hadoop system. The resource allocation problem is an NP-complete problem [38]; such a problem has received substantial attention in cloud computing.

Facebook fair scheduler [40] and Yahoo capacity scheduler [13] used multiple queues to achieve resource allocation through a policy in which each queue is assigned to a different number of resources. Within multiple queues can separate each task into several job types according to different properties saving in different job queues, and then assigned to *TaskTracker* from different job queues with suitable scheduler. Thus providing users with various queues can maximize resource utilization. However, users may have more resources than necessary, causing resources to be wasted. In this study, we investigated how resource utilization in Hadoop systems can be balanced in heterogenous computing environments such as clouds. Here, the term "heterogeneous" implies that the computing ability (e.g., the number of CPUs) of all nodes is different. In a Hadoop system, the workload of MapReduce jobs submitted by clients often differs: a job may be split into several tasks that achieve the same function, but involve managing different data blocks. When the default job scheduling policy of Hadoop is applied, the task of a single job usually runs on the same *TaskTracker*. When a *TaskTracker* executes the same task for a single job, it is limited by some resources even though the other resources remain idle. Therefore, if we consider that jobs can divided into CPU-bound and I/O-bound jobs according to Equation (1) defined in Section 3.1, 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* performs the same tasks. However, some *TaskTracker*s are assigned to perform CPU-bound jobs, which are bounded by the CPU resource, and the other *TaskTracker*s are specialized to perform I/O-bound jobs, which are bounded by the I/O resource.

To improve the performance and data locality, this study proposes a novel dynamic grouping integrated neighboring search (DGNS) algorithm, which considers the node ability on both the MapReduce and HDFS layers. According the node ability, we assigned different number of tasklots in each *TaskTracker* and different number of data blocks in each *DataNode*. The performance of the proposed algorithms was compared with that of the default FCFS job scheduling policies of Hadoop, and other related schedulers of DMR, JAS, JASL, and DJASL, which are described in Section 2.4. The experimental results indicated that the proposed algorithms improved both the performance and data locality of the aforementioned job scheduler.

The remainder of this paper is organized as follows. Section 2 describes the default Hadoop, DMR, JAS, and JASL schedulers. In addition, related work is discussed. Section 3 presents four phases of the proposed algorithm: job classification, ratio table creation, node grouping, and neighboring search of a task assignment. Section 4 presents experimental results obtained in the heterogeneous computing environments. Finally, Section 5 concludes the paper.

2 Background

2.1 Job Workloads

Rosti et al. [30] proposed that jobs can be classified according to the resources used; some jobs require a substantial amount of computation, whereas other jobs require numerous I/O resources. In this study, jobs were classified into two types according to the workload: (1) CPU-bound jobs and (2) I/O-bound jobs. Characterization and performance comparisons of CPU- and I/O-bound jobs have been provided in [21], [29], [37]. CPU- and I/O-bound jobs can be parallelized to balance resource utilization [30], [33].

2.2 Default Hadoop Scheduler

Hadoop supports the MapReduce programming model originally proposed by Google [8] and it is a convenient means for developing applications (e.g., parallel computing, job distributing, 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, the input data are split into several data blocks, of which the block size can be set by the user, and then parallelized using a map task. The second phase is the reduce phase. A map task is performed to generate the output data as the intermediate data after the map phase is complete, and these intermediate data are then received and the final result is produced.

The default job scheduling policy of Hadoop is FCFS and comprises 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): A job in the *Job Queue*is initialized using the *JobTracker*, which splits the job into numerous tasks.

Step 3 (Task assignment): When a *TaskTracker* periodically (3 seconds is the default setting) sends a *Heartbeat* to

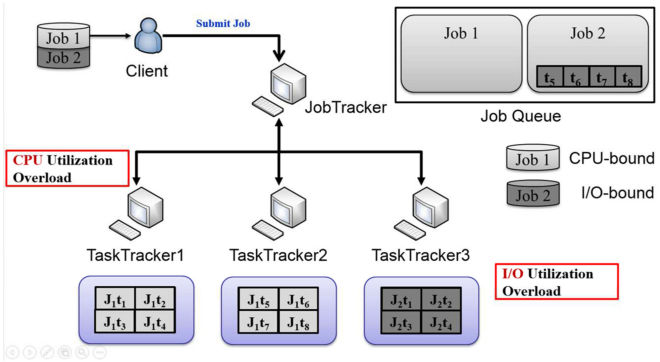


Fig. 1. Unbalanced resource allocation.

a *JobTracker*, the *JobTracker* obtains information related to the current state of the *TaskTracker* to determine whether it has available slots. If the *TaskTracker* has 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 schedulers of numerous parallel systems [4], [7], [22], [26], [27]. Specifically, the task schedulers, co-scheduler [4] and gang-scheduler [7], operate in a heterogeneous environment. The co-scheduler ensures that subtasks begin at the same time and are executed at the same pace on a group of workstations, whereas gang-scheduler uses a set of scheduled threads to execute tasks simultaneously on a set of processors. The task schedulers grid-scheduler [22], [26] and dynamic task scheduler [27], are designed to improve performance when the workloads of jobs differ. The grid-scheduler determines the order in which the jobs assigned to the distributed system are processed, and the dynamic task scheduler operates in an environment in which system resources vary and adapts to these variations. This study proposes a task scheduler that allocates resources at various job workloads in a heterogeneous computing environment.

2.3 Limitations of Default Hadoop Scheduler

The default job scheduling policy of Hadoop is FCFS, which can cause several problems including imbalanced resource allocation. Consider a situation in which numerous jobs are submitted and then split into many tasks that are assigned to *TaskTracker*s. Executing some of these tasks may require only CPU or I/O resources. Neglecting the workloads of a job may cause imbalanced resource allocation. Fig. 1 illustrates the imbalanced resource allocation in the default scheduler of Hadoop.

Assume that the *Job Queue* comprises two jobs: a CPU-bound job *Job*1 and an I/O-bound job *Job*2, which are initialized using eight map tasks. Moreover, *TaskTracker*1, *TaskTracker*2, and *TaskTracker*3 send *Heartbeat* messages to the *JobTracker* sequentially. 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. Conversely, *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 Other Schedulers

To address the imbalanced resource allocation problem of the default scheduler in Hadoop described in Section 2.3, Tian et al. [33] proposed a dynamic MapReduce (DMR) function for improving resource utilization during various types of jobs. However, the experimental environment of DMR is homogeneous. In a heterogeneous environment, the DMR function might not be favorable for resource utilization. To overcome the limitations of the current MapReduce application platforms, Hsieh et al. [14] first proposed 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) for testing the capability of each *TaskTracker* (represented by a capacity ratio). According to the capacity ratio for the two classes of jobs, *TaskTracker*s have different slots corresponding to the job types for maximizing resource usage. However, if tasks are assigned only according to the two job types, then it may not have the benefit of locality [12], increasing the data transfer time. The following three phases of the JAS algorithm are executed.

Step 1: Job classification. When jobs are in the *Job Queue*, the *JobTracker* does not know the type (CPU- or I/O-bound) of each job. Thus, the job types must be classified and the jobs must be added to the corresponding queue.

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

Step 3: Task assignment. When a *TaskTracker* sends a *Heartbeat* message, the *JobTracker* receives the number of idle CPU and I/O slots, and then assigns job tasks of various types to the corresponding slots for processing (i.e., the tasks of a CPU-bound job are assigned to CPU slots and those of a I/O-bound job are assigned to I/O slots).

Hadoop tends to assign tasks close to the node possessing its block; however, the JAS assigns tasks according to the number of slots, namely CPU and I/O slots, *Tasktracker*s possess. Because of the characteristic of the JAS, data locality is lost and the amount of network traffic is substantial. Fig. 2 illustrates the problem. To address the problem, the JAS was modified to ensure that the benefit of data locality is retained (Fig. 3). For considering the data locality, a table named *LocalityBenefitTable* is created to store the request time of each task [14]. The execution times of all *TaskTrackers* are stored into the *LocalityBenefitTable* in the cluster, including the times that the *TaskTrackers* require to execute local map task and non-local map tasks. And local map tasks

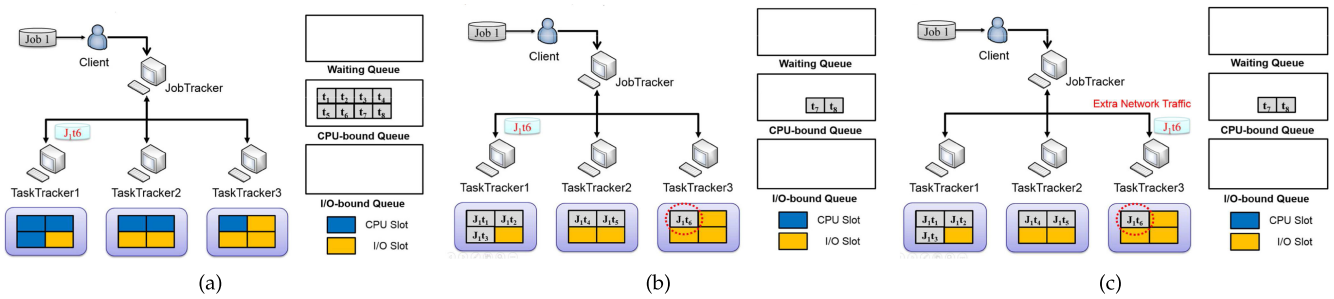


Fig. 2. Drawback of a JAS scheduler. (a) When a new job is submitted through a client and added to the CPU-bound queue, the JAS assigns tasks according to the number of slots to each *TaskTracker*. (b) According to the JAS, the following task must be assigned to *TaskTracker*3, which does not have the required data block. (c) Transferring the data from *TaskTracker*1 to *TaskTracker*3 causes extra network traffic.

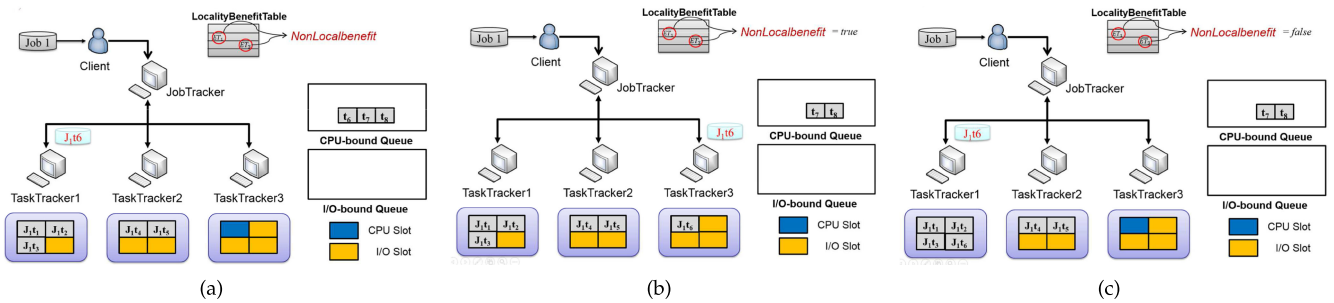


Fig. 3. Work flow of a JASL scheduler. (a) When a new job is submitted through a client, it is added to the waiting queue. The scheduler then predicts the job type. (b) After classifying the job type, the jobs are added to the CPU- or I/O-bound queue. The *JobTracker* then assigns these tasks according to the number of free CPU or I/O slots that the *TaskTracker* has. (c) *JobTracker* assigns the tasks until all of the *TaskTracker*s have no free slots.

means that the resources required by these tasks already on the located *TaskTracker*. The *JobTracker* then assigns either CPU- or I/O-bound tasks according to the *LocalBenefit* equation. If *NonLocalBenefit* is *true*, then the *TaskTracker* can execute a non-local task; otherwise, it can execute only local tasks.

To address this problem, a modified form of the JAS, called the JAS with locality (JASL), was developed [14]. The JASL can record the execution time of each node, and then compare the execution time 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 JASL (DJASL), was developed by adding a dynamic function to the JASL [14]. The aforementioned job scheduler improved the performance and data locality step by step. Although JASL and DJASL improved the data locality compared with JAS, their performance was below expectation.

2.5 Related Works

Because of the increase in the number of data, balancing resource utilization is a valuable method for improving the performance of a Hadoop system.[1] Recently numerous resource allocation algorithms have been proposed. Based on the analysis of the inadequacy of the chaos immune algorithm, Liang et al. [24] proposed an improved chaos immune algorithm for solving the job-shop problem. A combined chaos mapping was used to increase the diversity of the initial population and MapReduce was used to reduce the time complexity of the algorithm based on the Hadoop framework.

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

Zhang et al. [44] introduced PRISM, a fine-grained resource-aware MapReduce scheduler that divides tasks into phases, where each phase has a constant resource usage profile, and performs scheduling at the phase level. It first demonstrate the importance of phase-level scheduling by showing the resource usage variability within the lifetime of a task using a wide-range of MapReduce jobs. PRISM offers high resource utilization and provides 1.3x improvement in job running time compared to the current Hadoop schedulers.

Bezerra et al. [5] added a RAMDISK for temporary storage of intermediate data. A shared input policy schedules batches of data-intensive jobs that share the same input data set. 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 and Ramakrishnan [10] proposed a set of pipelining strategies to effectively utilize provisioned cloud resources. The experiments on the ExoGENI cloud testbed demonstrates the effectiveness of their approach in increasing performance and reducing failures.

Wang et al. [36] addressed the data locality problem from a stochastic network perspective. Focus on strike the right balance between data locality and load balancing to simultaneously maximize throughput and minimize delay. Authors present a new queueing architecture and propose a map task scheduling algorithm constituted by the Join the Shortest Queue policy together with the Max-Weight policy. The proposed algorithm is also delay-optimal in the heavy-traffic regime.

Wang et al. [35] used the round robin technique with a multiple feedback algorithm to solve this problem. Through

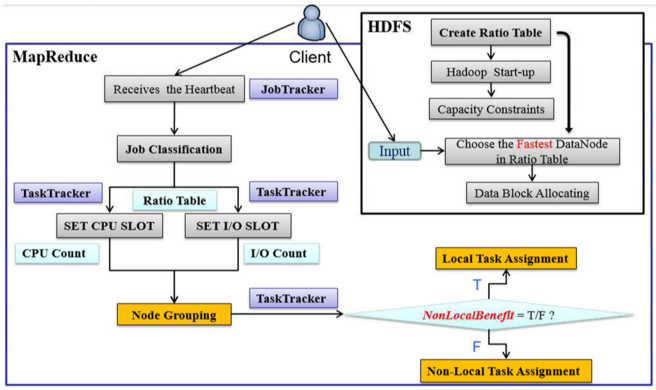


Fig. 4. Work flow of DGNS scheduler.

this scheduler, the job which is submitted late, receives quick response and is started without long delay. The results of the experiments on the Hadoop benchmark GridMix indicate that this algorithm can reduce the average response time by 10.

Isard et al. [16] mapped the problem to a graph data structure and then used a standard solver that computes the optimal online scheduler, which is called *Quincy*. When fairness is required, *Quincy* increases fairness and, thus, substantially improves data locality.

Tumanov et al. [34] focused on the resource allocation of mixed workloads in heterogeneous clouds and proposed an algebraic scheduling policy for mixed workloads in heterogeneous clods called *Alsched*, which allocates resources by regarding the composable utility functions submitted as resource requests.

Schwarzkopf et al. [31] 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 [31] to reach the flexibility for large computing clusters, revealing that optimistic concurrency over a shared state is a viable and attractive approach to cluster scheduling.

Max-min fairness is a resource allocation mechanism used frequently in most data center schedulers. However, numerous jobs are limited to specific hardware or software in the machines that execute them. Ghodsi et al. [9] proposed an offline resource allocation algorithm called *constrained max-min fairness* (CMMF), which is an extension of MMF and supports placement constraints. In addition, they also proposed an online version called *Choosy*. Apache has released a new version of the Hadoop system, called YARN or MapReduce2.0 (MRv2)[3], which is now adopted by Yahoo. The two major functions of *JobTracker*s, namely resource management and job scheduling and monitoring, are split into two components called *ResourceManager* and *NodeManager*. *ResourceManager* is used for scheduling the demand of resources that applications require, and *NodeManager* is used for monitoring 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 heterogeneous computing environments. Therefore, new scheduling algorithms/policies proposed in this paper are applicable to both Hadoop and YARN.

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 Proposed Algorithm

This section introduces the proposed job scheduler with dynamic grouping integrated neighboring search, which considers both the MapReduce and HDFS layers. The proposed scheduler provides each *TaskTracker* with a distinct number of slots according to the ability of the *TaskTracker* and each *DataNode* with different number of data blocks according to the ability of the *DataNode*. The proposed DGNS algorithm considers heterogeneous workloads and computing environments.

When a *TaskTracker* sends a *Heartbeat* message, the following four phases of the DGNS algorithm are executed (Fig. 4).

Phase1: Job classification.
When jobs are in the *Job Queue*, the *JobTracker*does not know the type(CPU- or I/O-bound) of each job. Thus, the job types must be classified and jobs must be added to the corresponding queue.

Phase2: Ratio table creation.
(phase2.1:) Create a capability ratio table of the *TaskTracker* (Map layer).
(phase2.2:) Set CPU and I/O slots (Map layer).
(phase2.3:) Create a capability ratio table of the *DataNode* (HDFS layer).

Phase3: Grouping and data block allocation.
(phase3.1:) Grouping with CPU slots number (Map Layer).
(phase3.2:) Capacity constraints and data block allocation (HDFS layer).

Phase4: Neighboring search.
(phase4.1:) CPU task allocation (Map layer).
(phase4.2:) I/O task allocation (Map layer).

3.1 Job Classification

Algorithm 1 is used for classifying jobs. When a *TaskTracker* sends a *Heartbeat* message, the *JobTracker* can determine the number of tasks that has been executed in the *TaskTracker*, and the *JobTracker* can determine the states of these tasks. When these tasks are complete, the JobTracker can receive the following information from the tasks (Table 1).

Refer to [33] for the parameters used to classify jobs (Table 1). Suppose that a *TaskTracker* has n slots and executes the same n map tasks, and the completion times of map tasks are the same. A map task generates data throughput including the execution time of map input data (MID), map output data (MOD), shuffle output data (SOD), and shuffle input data (SID); therefore, n map tasks generate the total data $throughput = n \times (MID + MOD + SOD + SID)$,

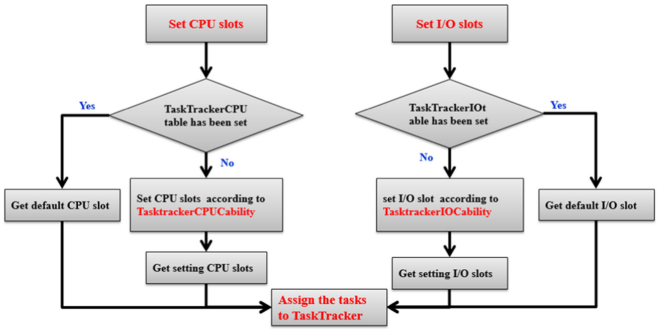


Fig. 5. Work flow of setting CPU slots.

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

JobTracker can determine how much data one map task can generate from the corresponding *TaskTracker* in 1 second which can be represented as throughput (Equation (1)). Then, the *JobTracker* determines the type of a job based on this information if *throughput* is less than disk average I/O rate (DIOR), in which case, the n map tasks generate total data throughput that is still less than the average I/O read/write rate. Any map task that requires I/O resources is small, causing the *JobTracker* to classify the job to which these map tasks belong as CPU-bound. However, if *throughput* is higher than or equal to *DIOR*, the n map tasks generate total data throughput that is higher than or equal to the average I/O read/write rate. In this case, the *JobTracker* classifies the job to which these map tasks belong as I/O-bound. After the type of each job has been determined, the *JobTracker* must record the execution time of all tasks, which can be used for computing the number of CPU-slots (I/O-slots) for each *TaskTracker*.

3.2 Ratio Table

This phase comprises some important steps. Before setting up the CPU- and I/O-slots of each *TaskTracker*, we must calculate the execution time of tasks and records in *TaskTrackerCPUCapability*. After creating the capability ratio table of the *TaskTracker* and setting up all CPU slots, we create another ratio table of the *DataNode* in the HDFS layer for determining the number of data blocks that will be allocated to each *DataNode* according to the storage capacity. Sections 3.2.1 and 3.2.2 explains these steps.

3.2.1 TaskTracker Capability and Slot Setting

In Algorithm 1, if a task belonging to a CPU-bound job is completed, then the *JobTracker* records the execution time of the task on *TaskTrackerCPUCapability*. If the *JobTracker* detects that the CPU slot of the *TaskTracker* has not been set, then the *JobTracker* executes Algorithm 2 to set the CPU-slot of the *TaskTracker*. In Algorithm 2, the *JobTracker* reads the execution time of each task in *TaskTrackerCPUCapability*, and uses (2) to compute the CPU capability of each *TaskTracker*. Finally, by using (3) to compute the CPU capability ratio of each *TaskTracker*, the *JobTracker* can determine the number of CPU slots in each *TaskTracker*.

Fig. 5 shows the work flow of setting slots. Algorithm 2 is the CPU slot setting algorithm for a *TaskTracker*. Table 2 lists the used parameters.

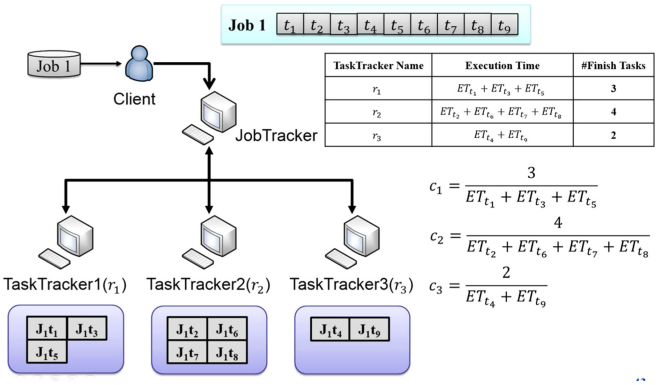
$$c_y = \frac{|M_y|}{e_y}; \quad (2)$$

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

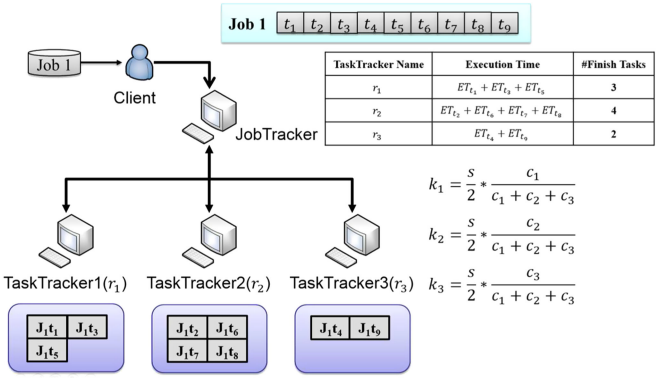
Fig. 6 illustrates the determination of the number of CPU slots in each *TaskTracker*. Assume that a client submits a job

TABLE 2
Parameters of 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 $t_i \in T$
$M_i = \{t_j \in T \mid t_j$ run 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 $TaskTracker_y$



(a) Calculate the CPU execution capability of the *TaskTracker*.



(b) Calculate the number of CPU slots in *TaskTracker*.

Fig. 6. CPU slot setting.

J to a *JobTracker*, where J is a CPU-bound job that has been divided into nine tasks. Three *TaskTracker*s r_1, r_2, and r_3 are present in this Hadoop system. The nine tasks are distributed by the *JobTracker* such that t_1, t_3, and t_5 are in r_1; t_2, t_6, t_7, and t_8 are in r_2; and t_4 and t_9 are in r_3. Whenever a task is completed, the *JobTracker* determines that the task belongs to a CPU-bound job and records the execution time of that task on *TaskTrackerCPUCapability*. For example, when t_1 is completed, the *JobTracker* determines that t_1 is a CPU-bound job and records the execution time of t_1 in r_1 on *TaskTrackerCPUCapability*, and then *TaskTrackerCPUCapability* determines that t_1 has been completed by r_1. By repeating the aforementioned steps, t_2–t_9 are recorded on *TaskTrackerCPUCapability*. Fig. 6 shows the results after determining all the *TaskTrackerCPUCapability*. After the job is complete, the *JobTracker* determines whether the CPU slot in the corresponding *TaskTracker* has been set to one; if yes, the *JobTracker* skips Algorithm 2; if not, the *JobTracker* executes Algorithm 2. According to *TaskTrackerCPUCapability*, the *JobTracker* uses (2) to compute the CPU capability of each TaskTracker. Equation (2) shows the number of tasks belonging to J on the *TaskTracker* that can be completed in 1 second. For example, the capacities of r_1, r_2, and r_3 are $c_1 = \frac{t_1+t_3+t_5}{3}$, $c_2 = \frac{t_2+t_6+t_7+t_8}{4}$, and $c_3 = \frac{t_4+t_9}{2}$, respectively. After determining the capacity of each *TaskTracker*, the *JobTracker* uses (3) to compute the CPU capability ratio of each *TaskTracker*, and further calculates the number of CPU slots of each *TaskTracker*. For example, the number of CPU slots on r_1, r_2, and r_3 are $k_1 = \frac{s}{2} \times \frac{c_1}{c_1+c_2+c_3}$, $k_2 = \frac{s}{2} \times \frac{c_2}{c_1+c_2+c_3}$, and $k_3 = \frac{s}{2} \times \frac{c_3}{c_1+c_2+c_3}$, respectively. Because the number of CPU slots (the 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 $\frac{s}{2}$.

TABLE 3
Ratio Table

	Node A	Node B	Node C
Job1	0.4	0.2	0.1
Job2	0.42	0.21	0.14
...	0.43	0.18	0.11

After executing Algorithm 2, the *JobTracker* can determine the number of CPU slots in each *TaskTracker*, improving the CPU resource utilization in each *TaskTracker*.

Algorithm 2. SET_CPU_SLOT(*Job Queue*)

```
for each Job in Job Queue do
  if Job is completed and CPU-bound then
    obtain the task information from the 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 / Σ_{j=1}^{m} c_j;
      record k_y on TaskTrackerCPUTable;
      SetTaskTrackerCPUTable := 1;
    return TaskTrackerCPUslot according to
    TaskTrackerCPUTable;
    break.
```

3.2.2 DataNode Capability

Before Hadoop startup, we create a RatioTable, which is used for determining the allocation ratio of data blocks in the nodes of the cluster. In other words, the RatioTable records are the ratios of storage capacity of each DataNode, which is in the heterogeneous cluster. Furthermore, according to [19], which is proposed by Yahoo!, 77 percent of the jobs are map-only and, 14 percent of the jobs are map-mostly; therefore approximately 91 percent of all jobs are map jobs, with only 9 percent of the jobs being reduce mostly. Because the number of map jobs is ten times higher than reduce jobs, from the engineering perspective, we can ignore reduce job. By doing so, we can simplify the problem and thus focus on the real data. Under this assumption, the method utilized to compute the ratio of the DataNode is not the same as that used in [39], which is testing different types of jobs on each DataNode. The proposed strategy is to record the execution time of each DataNode, calculating the average execution time of each DataNode, and then transform the average execution time into the storage capacity ratio. After performing the aforementioned computation, we record the ratio in the RatioTable.

Table 3 shows an example of RatioTable. There are three nodes in the cluster and the computing capability of each node is different. Node A is the fastest node, followed by node B; node C is the slowest node. According to our experiment, a situation in which B is the fastest node, followed by C and then A will not occur. Other similar situations will not occur as well. We assume that unless the job type changes, the computing capability of each node does

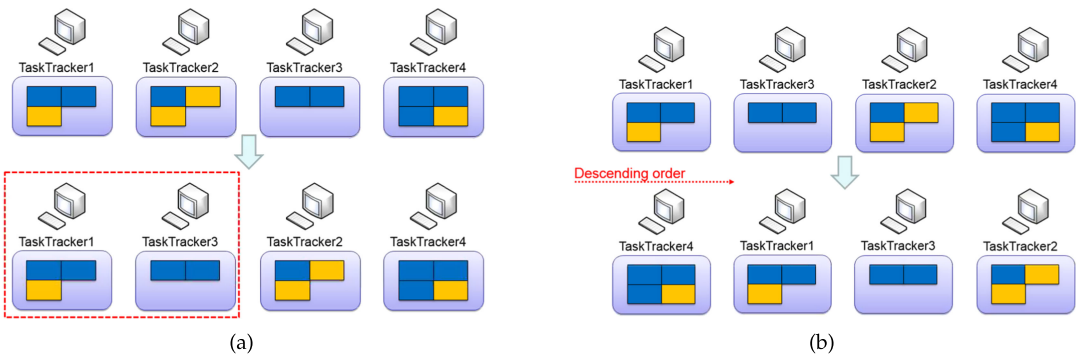


Fig. 7. Grouping with CPU slots. (a) Node grouping according to CPU slots number. (b) Node grouping with CPU slots number in descending order.

not change. Moreover, according to [19], 91 percent of all jobs are map jobs; therefore, we can ignore the scenario that computing capability of the DataNode may be changed because of the type of job.

Next, we describe the ratio computation. After the execution of each DataNode is completed, we divide the execution time of a slower node by that of the fastest node; by doing so, the computing ratio is obtained and we can transform and record it into the RatioTable. For example, in Table 3, Node A is the fastest node in the cluster; therefore, we divide the execution time of Node B by that of Node A and obtain the ratio of Node B=2, which implies that the execution time of Node B is two times slower than that of Node A. This also indicates that the storage capacity of Node A is two times greater than that of Node B. In other words, Node A processes more data than Node B, because the computing capability of Node A is higher than that of Node B.

However, each DataNode may comprise different number of task slots. The tasks in each task slot can be processed in parallel on DataNodes. This causes the ratio of execution, which was calculated based on the time of the task execution, to be inaccurate. Therefore, if we want to calculate the ratio of the storage capacity of each DataNode in accordance with the task execution time, the number of task slots must be considered.

Consider the following example. Assume that there are two DataNodes A and B. Node A is two times faster than node B, and the map slot numbers of nodes A and B are set to four and two, respectively. Assume that the execution times of the four tasks on each map slot of node A are 45, 43, 43, and 46 seconds and the four tasks execute simultaneously, the average execution time to complete one task is 44.25 seconds. To execute two tasks, node B requires 39 and 40 seconds; therefore, the average execution time is 39.5 seconds. If we calculate the computing ratio only in accordance with the average execution time of nodes, the value of node B is higher than that of node A. However, the actual computing efficiency of node A is higher than that of node B. Although the average execution time of node B is less than that of node A, the task slots assigned to node A and node B are different. Node A can execute four tasks simultaneously; however, node B only executes two tasks simultaneously. Therefore, the execution time must be divided by the number of task slots to obtain the average time required to complete one task. Let T_{avg} be the average execution time and S be the number of task slots. In addition, let T_t be the average time required to complete one task. Then,

$$T_t = \frac{T_{avg}}{S}. \tag{4}$$

According to the aforementioned example, the average time required by node A to complete one task is 11 seconds, and that required by node B equals 20 seconds. Therefore, the efficiency of node A is two times higher than that of node B. Therefore, we use T_t to calculate the computing capability of each DataNode, instead of directly dividing the execution time of a slower node by that of the fastest node. Finally, we transform the computing capability of each DataNode to the storage ratio, and record it into the RatioTable.

3.3 Grouping and Allocation

After classifying the types of jobs and setting up numbers of the slot to each *TaskTracker*, we use a new strategy, which groups *TaskTrackers* according to the CPU-slot numbers on each *TaskTracker*; the example is presented in Section 3.3.1. Moreover, after we create the ratio table of the *DataNode*, we allocate data blocks according to the storage ability of each *DataNode* from the start (fastest node). A simple example is presented in Section 3.3.2.

3.3.1 Grouping

Fig. 7a shows that there are four *TaskTracker*s with different number of CPU slots. *TaskTracker*1 has two CPU slots and one I/O slot, *TaskTracker*2 has one CPU slot and two I/O slots, *TaskTracker*3 has two CPU slots and zero I/O slot, *TaskTracker*4 has three CPU slots and one I/O slot. According to the CPU-slot number, we can place *TaskTracker*1 and *TaskTracker*3 in the same group, which means they possess the same execution ability for executing CPU-bound jobs. Fig. 7b shows the next step. We sort the *TaskTracker* in the descending order according to the group with different execution ability. We can update the number of *TaskTrackers* and node ability of each *TaskTracker* based on the capability ratio table, which we created in Section 3.2.1, to provide dynamic adjustment.

3.3.2 Data Block Allocation

In this section, we create a storage ratio table of each *DataNode*. According to this ratio table we assign each *DataNode* different capacities to allocate different number of data

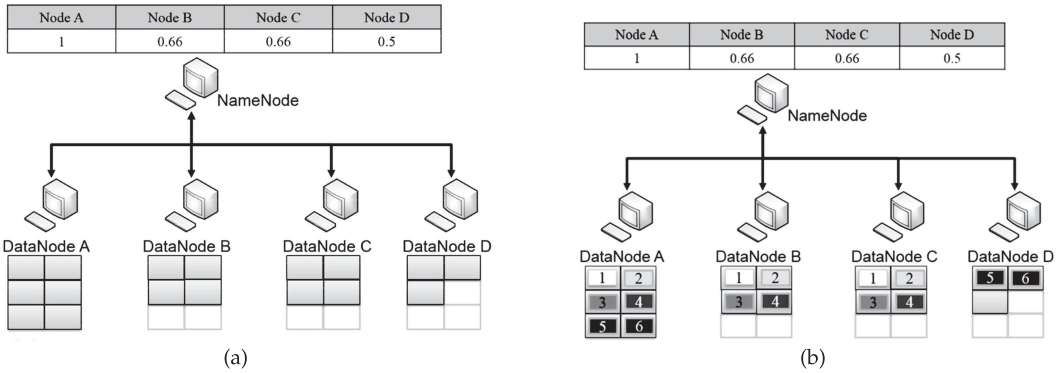


Fig. 8. Data block allocation. (a) Before data block allocated. (b) After data block allocated.

blocks. To improve performance, we select the fastest *DataNode* to allocate a data block first, and allocate same replica to *DataNode* in descending order according to node capacity. Moreover, we avoid allocating the replica of the same data block to the same *DataNode*. Algorithm 3 and Fig. 8 show a simple work flow to allocate the data blocks.

Algorithm 3. Allocat_Data_Block

1: *Nodename* := set by programmer;
2: *ratio* := get from *RatioTable* in accordance with *Nodename*;
3: *Selected Node* := the data that is allocated on;
4: **for** *Nodename* in the *RatioTable* **do**
5: *Compare* each *ratio* of *DataNode with RatioTable*;
6: **if** the ratio is the fastest in *RatioTable* **then**
7: *Selected Node* := *Nodename*;
8: *Allocating data block to the SelectedNode*.

3.4 Neighboring Search

Although MapReduce and HDFS are consider different layers of the Hadoop architecture, the *TaskTracker* and *DataNode* actually run on the same computing node. Fig. 9 shows the overview of the *TaskTracker* and *DataNode*.

After node grouping and data block allocation, assigning tasks to each *TaskTracker* is crucial. Fig. 10 shows the simple processes task assignment through neighboring search. Fig. 10a shows that there are three groups of *TaskTracker*s according to the CPU-slot numbers of each *TaskTracker*. When task3 of job3 has to be assigned, group1 comprising *TaskTracker*1, *TaskTracker*2, and *TaskTracker*3 is the first choice. In Fig. 10c, if the data block required by task3 of job3 is not present in the first group, we must consider the *NonLocalbenefit* to determine the *TaskTracker* that is the most favorable choice for reducing the additional traffic flow and exhibits higher performance. Fig. 10d illustrates an example that considers *NonLocalbenefit* according to

$$NonLocalBenefit = \begin{cases} true, & \text{if } ET_i < ET_j + TransferTime \\ false, & \text{otherwise.} \end{cases} \quad (5)$$

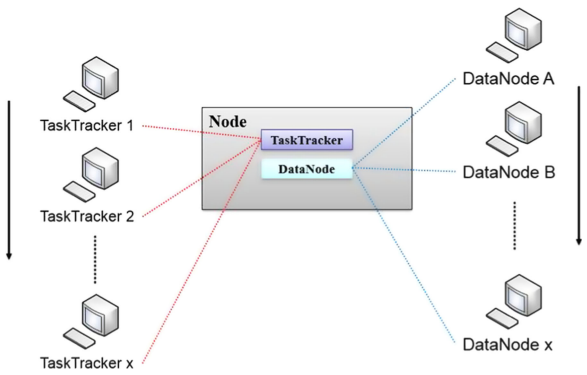


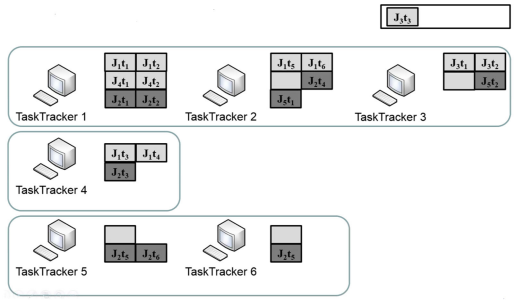
Fig. 9. Overview of the *TaskTracker* and *DataNode* in the computing node.

4 Performance Evaluation

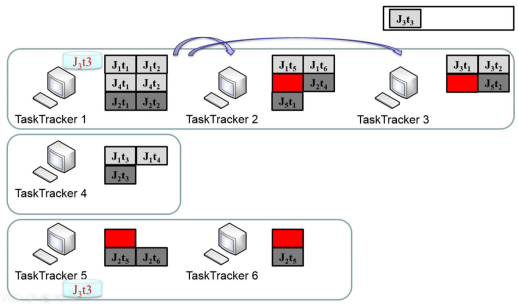
This section presents evaluation of proposed dynamic grouping integrated neighboring search algorithm on a Cloud experimental testbed environment created at National Cheng Kung University in Taiwan. We compare evaluation results of our algorithm with existing scheduling algorithms including default one present in Hadoop.

4.1 Experimental Environment

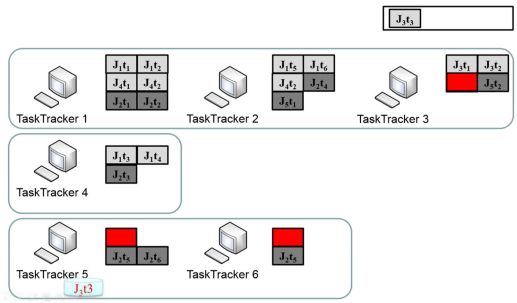
The experimental setup included IBM Blade Center H23 with seven blades (84 CPUs, 1450-GB memory, and a 3-TB disk partitioned into four spaces) and Synology DS214play NAS was mounted to extend hardware resources (Intel Atom CE5335 CPU and 3-TB disk space). Moreover, VirtualBox 4.2.8 was used to create several virtual machines. One of the machines was the master and the other machines were slaves. Several heterogeneous experimental environment setups shown in Tables 4 and 5 were employed to observe performance diversification and data locality. The first setup, Environment 1, comprised one master, which had two CPUs with 4-GB memory, and 99 slaves, each having one CPU with 2-GB memory. Environment 1 was set to the control group. The second setup, Environment 2 comprised one master, which had four CPUs with 16-GB memory, and 99 slaves, each having one CPU with 2-GB memory. Environment 2 was set up for considering the different computing abilities of the master node. The third setup, Environment 3, comprised one master, which had two CPUs with 4-GB memory, and 99 slaves. Among these 99 slaves, 33 slaves had one CPU each with 2-GB memory, another 33 slaves had two CPUs each with 2-GB memory, and the remaining 33 slaves had four CPUs each with 2-GB memory. Environment 3 was set up for considering the different number of CPUs for different groups of slaves. The last setup, Environment 4, comprised one master, which had two CPUs with 4-GB memory, and 99 slaves. Among these 99 slaves, 33 slaves had one CPU



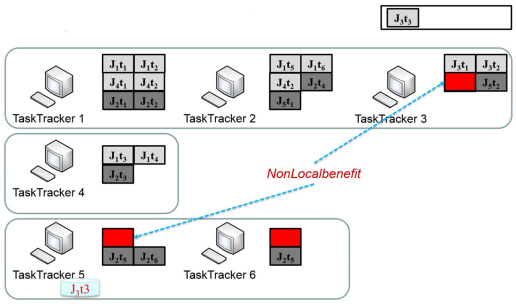
(a) *Tasktrackers* have been classified after node grouping.



(b) Assign task according to group ability and data block first.



(c) Data block located in poor group.



(d) Performance improvement using *NonLocalbenefit* for determining the most favorable choice.

Fig. 10. Task assignment through neighboring search.

each with 2-GB memory, another 33 slaves had one CPU each with 4-GB memory, and the remaining 33 slaves had one CPU each with 8-GB memory. Environment 4 was set up for considering the different amount of memory on different group of slaves. All machines shared 6-TB hard disk space. Ubuntu 14.04 LTS was adopted as the operating system. Hadoop Version Hadoop-0.20.205.0 was used, and each node had four map slots and one reduce slot.

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

TABLE 4
Heterogeneous Experimental Environment of the Master

	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	4 cpu & 16 GB memory	33	1 cpu & 2 GB memory
			33	1 cpu & 2 GB memory
			33	1 cpu & 2 GB memory

TABLE 5
Heterogeneous Experimental Environment of the CPU

	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	2 cpu & 2 GB memory
			33	4 cpu & 2 GB memory

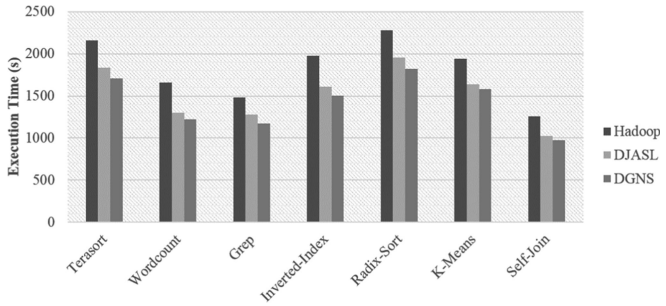


Fig. 11. Average execution time of each job with Hadoop, DJASL, and DGNS algorithms.

4.2 Results

The experimental results of this study can be classified as follows: (1) the individual performance of each job and the effect of different data sizes (Section 4.2.1); (2) the proposed DGNS algorithm improves the overall performance of the Hadoop system in different heterogeneous computing environments (Section 4.2.2); (3) the proposed DGNS algorithm improves the overall performance of the Hadoop system and the data locality is similar to that of the DJASL algorithm (Section 4.2.3).

4.2.1 *Individual Performance of Each Workload*

Fig. 11 illustrates the individual performance of each jobs and each job has a data size of almost 10 GB with 100 nodes in Environment 1. The average execution time of DGNS compared with the default Hadoop and DJASL algorithms in Environment 1 (Table 4) shows that sorting jobs require

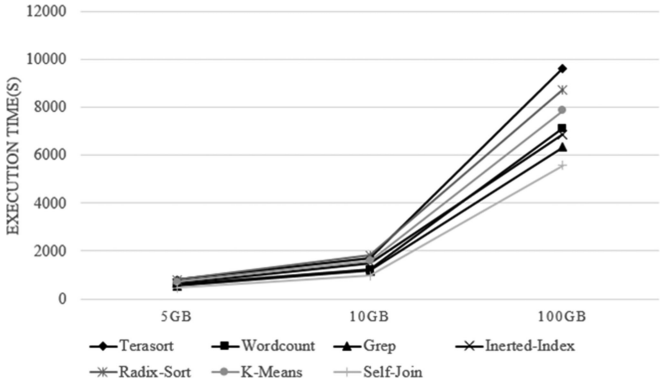


Fig. 12. Average execution time of each job with different data sizes.

more execution time than the others, and join jobs require less execution time. However, the results in Fig. 12 demonstrate that when multiple amounts of data are added, the average execution time does not significantly increases during the experiment. We designed almost 100 GB data size of each request comprising different jobs and performed batch processing by using DGNS algorithm. The experimental results are presented in the following sections.

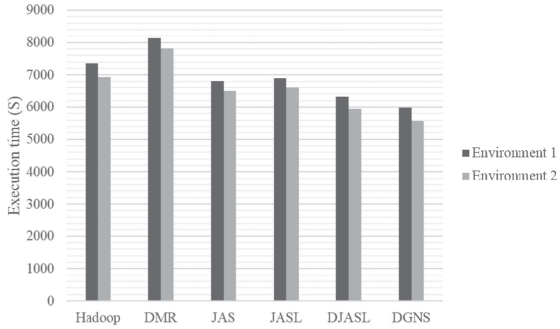
4.2.2 DGNS in Heterogeneous Computing Environments

We used three types of heterogeneous computing environments (Tables 4, 5, and 6) and compared each of them in detail with all algorithms presented in Section 2.5 (e.g., Hadoop, DMR, JAS, JASL, and DJASL). Fig. 13a shows the performance of the DGNS in Environment 2, which involved more CPUs and memory on the master node, compared with Environment 1. Fig. 13b shows the performance of the DGNS in Environment 3, which included more CPUs on the portion of slaves, compared with that of Environment 1. Fig. 13c shows the performance of the DGNS in Environment 4, which included more memory on the portion of slaves, compared with that of Environment 1. A comparison of these figures reveals that the number of CPUs highly affects the performance with respect to the amount of memory and higher ability to ability of the master node (Fig. 13). Moreover, the proposed algorithm (DGNS) has higher performance in these three heterogeneous computing environments.

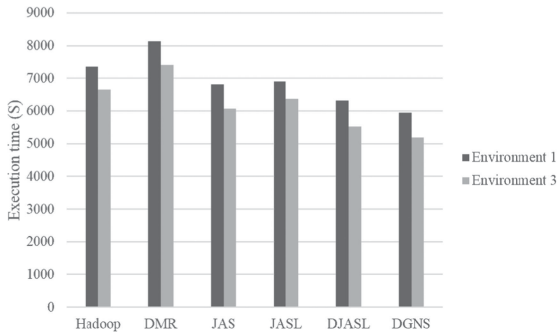
In some of the ten requests, the performance of the JAS was not more favorable than that of Hadoop and DMR because the JAS set slots incorrectly. Therefore, the resource utilizations of some *TaskTracker*s overloaded, and some tasks could not be executed until resources were released.

TABLE 6
Heterogeneous Experimental Environment of RAM

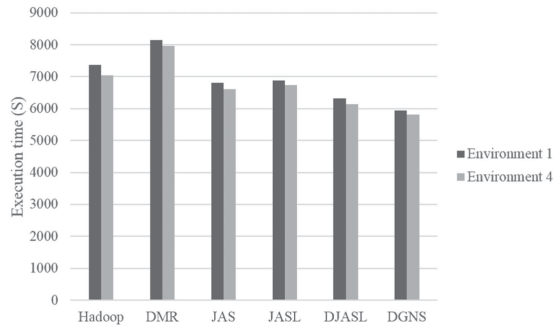
	Master		Slave	
	Quantity	Specification	Quantity	Specification
Environment 1	1	2 cpu & 4 GB memory	33	1 cpu & 2 GB memory
			33	1 cpu & 2 GB memory
			33	1 cpu & 2 GB memory
Environment 4	1	2 cpu & 4 GB memory	33	1 cpu & 2 GB memory
			33	1 cpu & 4 GB memory
			33	1 cpu & 8 GB memory



(a) Average execution time of DGNS in heterogeneous environment based on different abilities of the master node.



(b) Average execution time of DGNS in a heterogeneous environment based on different numbers of CPUs.



(c) Average execution time of DGNS in heterogeneous environment based on different amount of memory.

Fig. 13. Performance of DGNS compared with that of DJASL, JASL, JAS, DMR, and Hadoop in different heterogeneous computing environments.

Therefore, the execution times of these tasks increased, causing the performance of the JAS to decrease compared with that of Hadoop and DMR. However, for simulating real situations, the execution time of all jobs was averaged over ten requests. The average performance of the JAS and JASL was superior to that of Hadoop and DMR in a heterogeneous computing environment.

In the four heterogeneous computing environments, the DGNS algorithm improved performance by almost 21 percent compared with Hadoop and almost 8–10 percent compared with DJASL. Moreover, the DGNS algorithm improved the data locality in these four environments. The results of data locality are presented in Section 4.2.3.

4.2.3 Performance and Data Locality of DGNS

The proposed (DGNS) job scheduler improves the performance by considering the ability of each *TaskTracker*.

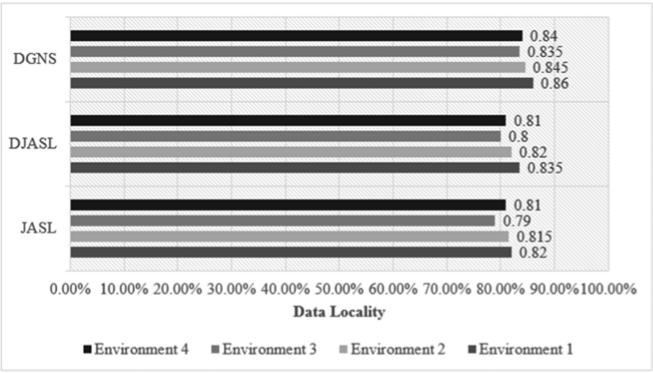


Fig. 14. DGNS data locality in heterogeneous computing environments.

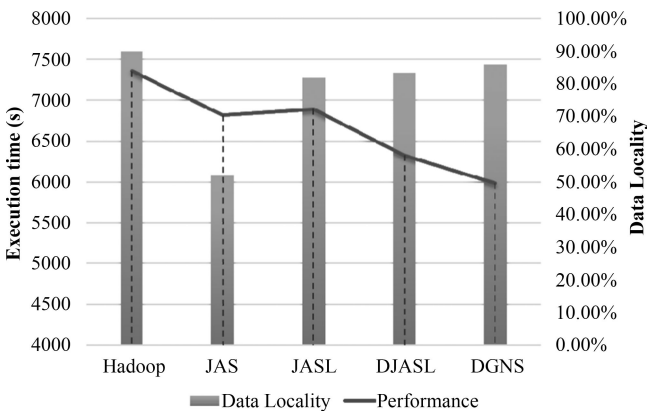


Fig. 15. Performance and data locality overview of the DGNS algorithm.

Moreover, it considers the ability of each *NameNode* by using the *NonLocalbenefit* equation for determining the most favorable choice to obtain considerable improvement.

Because the main DGNS strategy proposes that effective nodes not only have more slots to compute tasks but also have more resources, its performance on average is superior to that of Hadoop, DMR, JAS, JASL, and DJASL. In some scenarios, tasks that have been executed by *TaskTracker*s are not removed using the *JobTracker*. Therefore, the *JobTracker* must wait for these tasks to be completed. When *TaskTracker*s are overloaded, the tasks in these *TaskTracker*s cannot be completed until resources are released. Therefore, the execution times for these tasks are prolonged, causing performance to be poor compared with that of DMR. When the slots of the *JobTracker* are reset, the resources of each *TaskTracker* can be used appropriately to improve the performance of the Hadoop system. The average execution time of all jobs was used for simulating real situations.

As shown in Fig. 14, the proposed (DGNS) algorithm has higher data locality than JASL and DJASL in the four heterogeneous computing environments. Fig. 15 presents an overview of both performance and data locality. The results show that the DGNS algorithm not only effectively reduces the execution time but also improves the data locality compared with the default Hadoop, JAS, JASL, and DJASL algorithms.

5 Conclusion and Future Work

This paper proposed a new job scheduling algorithm, called DGNS, for providing a more efficient job scheduler for the Hadoop system. The default job scheduling of Hadoop does not consider the type of jobs, causing resource utilization in some *TaskTracker*s to be overloaded. In a previous study, the JAS, JASL, and DJASL algorithms improved the performance and data locality step by step. Although the DJASL algorithm effectively improved the data locality, it did not improve the performance as expected. Therefore, the DGNS proposed in this paper not only achieves higher resource utilization for each *TaskTracker* but also improves performance and data locality. The experimental results demonstrated that the proposed DGNS algorithm can improve the performance of the Hadoop system by approximately 21 percent, and approximately 8–10 percent compared with the DJASL. Moreover, it improved the data locality to 86 percent. In the future, the authors plan to modify this algorithm to focus on various types of jobs, maximizing resource utilization, and providing large scale experimental environments in the Hadoop system.

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.

References

[1] F. Ahmad, S. T. Chakradhar, A. Raghunathan, and T. N. Vijaykumar, "Tarazu: Optimizing MapReduce on heterogeneous clusters," *ACM SIGARCH Comput. Archit. News*, vol. 40, pp. 61–74, Mar. 2012.

[2] Apache Hadoop, 2014. [Online]. Available: http://hadoop.apache.org/

[3] Apache Hadoop YARN, 2008. [Online]. Available: http://hadoop.apache.org/docs/current/hadoop-yarn/hadoop-yarn-site/YARN.html

[4] 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.

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

[6] Q. Chen and Q. Deng, "Cloud computing and its key techniques," *J. Comput. Appl.*, vol. 29, pp. 2562–2567, 2012.

[7] D. G. Feitelson and L. Rudolph, "Gang scheduling performance benefits for fine-grained synchronization," *J. Parallel Distrib. Comput.*, vol. 16, pp. 306–318, 1992.

[8] S. Ghemawat, H. Gobioff, and S.-T. Leung, "The Google file system," *ACM SIGOPS Operating Syst. Rev.*, vol. 37, pp. 29–43, Dec. 2003.

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

[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] R. Gu, et al., "SHadoop: Improving MapReduce performance by optimizing job execution mechanism in Hadoop clusters," *J. Parallel Distrib. Comput.*, vol. 74, pp. 2166–2179, Mar. 2014.

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

[13] Hadoop's Capacity Scheduler, 2013. [Online]. Available: https://hadoop.apache.org/docs/r1.2.1/capacity_scheduler.html

[14] S. Y. Hsieh, C. T. Chen, C. H. Chen, T. G. Yen, H. C. Hsiao, and R. Buyya, "Novel scheduling algorithms for efficient deployment of MapReduce applications in heterogeneous computing environments," *IEEE Trans. Cloud Comput.*, doi: 10.1109/TCC.2016.2552518.

[15] S. Ibrahim, H. Jin, L. Lu, S. Wu, B. He, and L. Qi, "LEEN: Locality/fairness-aware key partitioning for MapReduce in the cloud," in *Proc. IEEE 2nd Int. Conf. Cloud Comput. Technol. Sci.*, 2010, pp. 17–24.
[16] 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.
[17] *ITU Focus Group on Cloud Computing - Part 1, Telecommunication Standardization Sector of the International Telecommunications Union. Retrieved* Dec. 16, 2012.
[18] Y. Jadeja and K. Modi, "Cloud computing-concepts, architecture and challenges," in *Proc. Int. Conf. Comput. Electron. Elect. Technol.*, 2012, pp. 877–880.
[19] S. Kavulya, J. Tan, R. Gandhi, and P. Narasimhan, "An analysis of traces from a production MapReduce cluster," in *Proc. 10th IEEE/ACM Int. Conf. Cluster Cloud Grid Comput.*, 2010, pp. 94–103.
[20] G. Kousiouris, T. Cucinotta, and T. Varvarigou, "The effects of scheduling, workload type and consolidation scenarios on virtual machine performance and their prediction through optimized artificial neural networks," *J. Syst. Softw.*, vol. 84, pp. 1270–1291, Aug. 2011.
[21] 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.
[22] H. Y. Lee, D. W. 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.*, 2006, pp. 53–62.
[23] B. Li, et al., "Cloud manufacturing: A new service-oriented networked manufacturing model," *Comput. Integr. Manuf. Syst.*, vol. 16, pp. 1–7, 2010.
[24] X. Liang, M. Wang, X. Jiao, and M. Huang, "An improved chaos immune algorithm based on Hadoop framework to solve job-shop scheduling problem," in *Proc. 3rd Int. Conf. Comput. Sci. Netw. Technol.*, 2013, pp. 5–9.
[25] *Network Virtualisation–Opportunities and Challenges*, Eurescom, Retrieved Dec. 16, 2012.
[26] J. K. Ousterhout, "Scheduling techniques for concurrent systems," in *Proc. 3rd Int. Conf. Distrib. Comput. Syst.*, 1982, pp. 22–30.
[27] 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, pp. 189a–189a.
[28] 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, Art. no. 7.
[29] 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, pp. 56–65, Jun. 1998.
[30] 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.
[31] M. Schwarzkopf, A. Konwinski, M. Abd-El-Malek, and J. Wilkes, "Omega: Flexible, scalable schedulers for large compute clusters," in *Proc. 8th ACM Eur. Conf. Comput. Syst.*, 2013, pp. 351–364.
[32] *The role of virtualisation in future network architectures, Change Project*. Retrieved Dec. 16, 2012.
[33] C. Tian, H. Zhou, Y. He, and L. Zha, "A dynamic MapReduce scheduler for heterogeneous workloads," in *Proc. 8th Int. Conf. Grid Cooperative Comput.*, 2009, pp. 218–224.
[34] 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, Art. no. 25.
[35] Y. Wang, R. Rao, and W. Wang, "A round robin with multiple feedback job scheduler in Hadoop," in *Proc. IEEE Int. Conf. Progress Informat. Comput.*, 2014, pp. 471–475.
[36] W. Wang, K. Zhu, L. Ying, J. Tan, and L. Zhang, "MapTask scheduling in MapReduce with data locality: Throughput and heavy-traffic optimality," *IEEE/ACM Trans. Netw.*, vol. 24, no. 1, pp. 190–203, Feb. 2016.
[37] Y. Wiseman and D. G. Feitelson, "Paired gang scheduling," *IEEE Trans. Parallel Distrib. Syst.*, vol. 14, no. 6, pp. 581–592, Jun. 2003.
[38] J. Weinman, "Cloud Computing is NP-Complete" Working Paper, 2011.
[39] J. Xie, et al., "Improving MapReduce performance through data placement in heterogeneous Hadoop clusters," in *Proc. IEEE Int. Symp. Parallel Distrib. Process. Workshops PhD Forum*, 2010, pp. 1–9.
[40] M. Zaharia, A. Konwinski, A. D. Joseph, R. Katz, and I. Stoica, "Improving MapReduce performance in heterogeneous environments," in *Proc. 8th USENIX Conf. Operating Syst. Design and Implementation*, pp. 29–42, 2008.
[41] M. Zaharia, D. Borthakur, J. Sen 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.
[42] Q. Zhang, L. Cheng, and R. Boutaba, "Cloud computing: State-of-the-art and research challenges," *J. Internet Serv. Appl.*, vol. 1, pp. 7–18, 2010.
[43] 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.
[44] Q. Zhang, M. F. Zhani, Y. Yang, R. Boutaba, and B. Wong, "PRISM: Fine-grained resource-aware scheduling for MapReduce," *IEEE Trans. Cloud Comput.*, vol. 3, no. 2, pp. 182–194, Apr.–Jun. 2015.

Chi-Ting Chen received the MS degree from the Department of Computer Science and Information Engineering, National Cheng Kung University, Tainan, Taiwan, in July 2015. He then studied PhD degree for two semesters from National Cheng Kung University from February 2016 to January 2017. He has published one cloud computing papers in the *IEEE Transactions on Cloud Computing* in 2016. Now he is going to serve the research and development substitute services for three years. His research interests include hub location problem, cloud computing, distributed computing, and design and analysis of algorithms.

Ling-Ju Hung received the bachelor's degree in applied mathematics from National Chung Hsing University, Taiwan, in 2001 and the MS and PhD degrees in computer science and information engineering from National Chung Cheng University, Taiwan, in 2003 and 2012, respectively. She became a postdoctoral research fellow in the Department of Computer Science and Information Engineering, Hung Kuang University, Taiwan during 2012–2015. She is currently a postdoctoral research fellow in the Department of Computer Science and Information Engineering, National Cheng Kung University. Her research interests include the design and analysis of fixed-parameter algorithms, exact algorithms, approximation algorithms, and graph theory.

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 in 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 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.

Rajkumar Buyya is professor of Computer Science and Software Engineering, future fellow of the Australian Research Council, and director of the Cloud Computing and Distributed Systems (CLOUDS) Laboratory, 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 four text books including "*Mastering Cloud Computing*" published by McGraw Hill, China Machine Press, and Morgan Kaufmann for Indian, Chinese and international markets respectively. He also edited several books including "*Cloud Computing: Principles and Paradigms*" (Wiley Press, USA, Feb. 2011). He is one of the highly cited authors in computer science and software engineering worldwide (h-index = 108, g-index = 225, 55800+ citations). Microsoft Academic Search Index ranked as the world's top author in distributed and parallel computing between 2007 and 2015. "*A Scientometric Analysis of Cloud Computing Literature*" by German scientists ranked as the World's Top-Cited Author and the World's Most-Productive Author in Cloud Computing. 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. His contributions and international research leadership 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 served as the foundation editor-in-chief of the *IEEE Transactions on Cloud Computing*. He is currently serving as co-editor-in-chief of the *Journal of Software: Practice and Experience*, which was established 40+ years ago. He is a fellow of the IEEE. For further information please visit his cyberhome: www.buyya.com.

Albert Y. Zomaya (M90-SM97-F04) is currently the chair professor of high performance computing & networking in the School of Information Technologies, University of Sydney. He is also the director of the Centre for Distributed and High Performance Computing which was established in late 2009. He was an Australian Research Council Professorial fellow during 2010-2014. He published more than 500 scientific papers and articles and is author, co-author or editor of more than 20 books. He is the founding editor in chief of the *IEEE Transactions on Sustainable Computing* and previously he served as editor in chief for the *IEEE Transactions on Computers* (2011-2014). Currently, he serves as an associate editor for 22 leading journals, such as, the *ACM Computing Surveys*, the *IEEE Transactions on Computational Social Systems*, the *IEEE Transactions on Cloud Computing*, and the *Journal of Parallel and Distributed Computing*. He delivered more than 170 keynote addresses, invited seminars, and media briefings and has been actively involved, in a variety of capacities, in the organization of more than 700 national and international conferences. He is the recipient of the IEEE Technical Committee on Parallel Processing Outstanding Service Award (2011), the IEEE Technical Committee on Scalable Computing Medal for Excellence in Scalable Computing (2011), and the IEEE Computer Society Technical Achievement Award (2014). He is a chartered engineer, a fellow of the AAAS, the IEEE, the IET (UK), and an IEEE Computer Societys Golden Core member. His research interests lie in parallel and distributed computing, networking, and complex systems.