applied sciences

MDPI

Article

Minimizing Resource Waste in Heterogeneous Resource Allocation for Data Stream Processing on Clouds

Wu-Chun Chung [1], Tsung-Lin Wu [2], Yi-Hsuan Lee [2], Kuo-Chan Huang [2], Hung-Chang Hsiao [3] and Kuan-Chou Lai [2,*]

[1] Information and Computer Engineering, Chung Yuan Christian University, Taoyuan 320, Taiwan; wcchung@cycu.edu.tw
[2] Computer Science, National Taichung University of Education, Taichung 403, Taiwan; bcs105105@gm.ntcu.edu.tw (T.-L.W.); yslee@mail.ntcu.edu.tw (Y.-H.L.); kchuang@mail.ntcu.edu.tw (K.-C.H.)
[3] Computer Science and Information Engineering, National Cheng Kung University, Tainan 701, Taiwan; hchsiao@csie.ncku.edu.tw
* Correspondence: kclai@mail.ntcu.edu.tw

Abstract: Resource allocation is vital for improving system performance in big data processing. The resource demand for various applications can be heterogeneous in cloud computing. Therefore, a resource gap occurs while some resource capacities are exhausted and other resource capacities on the same server are still available. This phenomenon is more apparent when the computing resources are more heterogeneous. Previous resource-allocation algorithms paid limited attention to this situation. When such an algorithm is applied to a server with heterogeneous resources, resource allocation may result in considerable resource wastage for the available but unused resources. To reduce resource wastage, a resource-allocation algorithm, called the minimizing resource gap (MRG) algorithm, for heterogeneous resources is proposed in this study. In MRG, the gap between resource usages for each server in cloud computing and the resource demands among various applications are considered. When an application is launched, MRG calculates resource usage and allocates resources to the server with the minimized usage gap to reduce the amount of available but unused resources. To demonstrate MRG performance, the MRG algorithm was implemented in Apache Spark. CPU- and memory-intensive applications were applied as benchmarks with different resource demands. Experimental results proved the superiority of the proposed MRG approach for improving the system utilization to reduce the overall completion time by up to 24.7% for heterogeneous servers in cloud computing.

Keywords: cloud computing; big data; spark; resource allocation

Citation: Chung, W.-C.; Wu, T.-L.; Lee, Y.-H.; Huang, K.-C.; Hsiao, H.-C.; Lai, K.-C. Minimizing Resource Waste in Heterogeneous Resource Allocation for Data Stream Processing on Clouds. *Appl. Sci.* **2021**, *11*, 149. https://dx.doi.org/10.3390/app11010149

Received: 29 October 2020
Accepted: 22 December 2020
Published: 25 December 2020

1. Introduction

According to the Gartner report [1], the analysts forecast there are 5.8 billion Internet of Things (IoT) endpoints in 2020, which is increasing by 21% since 2019. The larger number of IoT devices generates a huge amount of data to the cloud. An efficient big data processing platform for the data streaming application is getting more attentions. Apache Hadoop [2] and Apache Spark [3] are two popular solutions to deal with the big data processing. Hadoop is an open-source implementation for MapReduce framework [4] while Spark is an in-memory processing framework for performance efficiency. The literature [5,6] studied the performance of Hadoop and Spark to demonstrate that the speedup of Spark is better than that of using Hadoop.

On the other hand, with the advancement of virtualization technology, most hardware resources such as CPU, memory, disk, and network I/O can be virtualized and shared in a modern cloud infrastructure. These virtualized resources function as a resource-provisioning pool and are dynamically provisioned according to the application demands. From the viewpoint of users, the allocation and management of virtualized resources is

provided by cloud services. However, the physical resources in the cloud infrastructure have to be administrated by the system. Therefore, the allocation of virtualized resources among physical servers is a crucial research topic in cloud computing. This paper focuses on tackling the essential problem of resource allocation and attempts to reduce the amount of available but unused resources in the cloud infrastructure.

Yousafzai et al. [7] address the overlapping concept among resource provisioning, resource allocation, and resource scheduling. In general, resource provisioning is the allocation process of resources in service-providers to customers. Resource allocation is the process of distributing resources efficiently to competing jobs. Resource scheduling is to obtain the time-schedule of resources to schedule computational events on shared resources in available time. The same description also could be found in the previous work [8].

A resource-allocation mechanism [9,10] plays a crucial role in determining an efficient strategy for allocating resources to satisfy application demands. However, the resource demand for various applications can be heterogeneous in cloud computing. For example, computing-intensive applications require more CPU resources for the computation task, whereas memory-intensive applications require more memory resources for the data cache. The effectiveness of a resource-allocation mechanism affects system performance. Over-provisioning and under-provisioning are two common problems for resource allocation in cloud computing [11]. Over-provisioning with excessive resources for application demands leads to lower resource utilization and higher capital expenditure. However, under-provisioning with fewer resources results in a lower resource capacity and a higher response time, which result in loss of users and revenue. Accordingly, a trade-off exists between the provision of resource capacities and the consumption of application demands. An efficient resource-allocation mechanism can optimize the allocation of resource capacities to satisfy application demands and improve overall resource utilization.

Many resource-allocation algorithms [12–19] have been proposed to investigate the resource-allocation problem on cloud computing. Some well-known resource management systems, such as Yarn [20] and Mesos [21], have been developed. Previous studies have focused on the computing environment with homogeneous resources. However, the development of a practical cloud computing environment with heterogeneous computing resources has received limited attention [22]. Resource heterogeneity is a common occurrence in a practical cloud system because various computing servers have different resource capacities. Some servers have more CPU resources and some of them may have more capacities of memory. In addition, resource heterogeneity in a hybrid cloud is significant because hardware equipment and resource capacity are heterogeneous between private and public clouds. Edge computing [23–26] is an ongoing paradigm shift in which resource types are more heterogeneous among geographical edge locations. The experimental results of our study indicate that the heterogeneity of resource capacities should be considered in the design of heterogeneous resource allocation as it reduces the amount of available but unused resources.

Compared to the literature, our contribution is to facilitate appropriate utilization of heterogeneous resources in cloud computing. A novel resource-allocation mechanism, named minimizing resource gap (MRG) algorithm, was proposed for solving the resource wastage problem for the available but unused resources. A resource gap is a phenomenon in which some resource capacities are exhausted, whereas other resources on the same server are still available. In this case, the computing server may not be able to satisfy the resource capacity demands from an application, which leads to a low resource utilization and resource wastage for the available but unused resources. In addition, the resource gap is more apparent when the computing resources are more heterogeneous. Therefore, the MRG algorithm was proposed in this study for reducing the resource gap by considering distinct resource demands in each application. For example, some applications require more CPU resources for computing-intensive tasks, whereas others require more memory capacity for memory-intensive processing. When a new application is submitted to the

computing system, MRG calculates the resource gap among servers and discovers the server with a minimal resource gap for allocating the application to the server. Thus, by reducing the resource gap in servers, the MRG algorithm can improve the resource utilization of computing servers for heterogeneous resource allocation.

MRG was implemented in Apache Spark for performance evaluation. Spark [27] is an open-source distributed computing framework written using Scala and has been a popular cloud computing platform adopted by many companies for big data [28]. Apache Spark allocates multiple servers in a cluster to solve the resource gap problem with large amounts of data volumes and computations for parallel and distributed processing. The default resource-allocation algorithm in Apache Spark can result in resource wastage because of a large resource gap in case of servers with heterogeneous resource capacities. To evaluate the performance of the proposed MRG approach, the performance of the default algorithm in Apache Spark is compared with our approach.

Two experiments were conducted in this study. In the first experiment, the improvement in resource allocation with the use of MRG in case of homogenous resource demands of applications was evaluated. The experimental results revealed that when the homogeneity of resource demand of applications was close to the homogeneity of server resources, the completion time of jobs by the MRG algorithm was considerably lower than that by the default algorithm in Spark. In the second experiment, the improvement by MRG was confirmed. The results revealed that when the homogeneity of server resources was lower than that of the application demand, MRG improved resource utilization and reduced the job completion time. Therefore, experimental results demonstrated that the proposed MRG approach outperforms default resource allocation in Apache Spark when resources are heterogeneous.

The rest of this study is organized as follows. The most relevant studies are discussed in Section 2. The system framework and algorithm are proposed in Section 3. Performance evaluations and experimental results are presented in Section 4. Finally, conclusion remarks and future studies are presented in Section 5.

2. Related Works

An efficient resource-allocation strategy to satisfy application demands is a vital topic in cloud computing. Shakarami et al. [29] proposed a systematic and detailed survey on stochastic-based offloading mechanisms. Arun and Prabu [30] presented a survey of resource-allocation approaches in mobile cloud computing. Yousafzai et al. [7] introduced a thematic taxonomy of the resource-allocation approaches and provided their strengths and weaknesses. Manvi and Shyam [8] introduced a survey of the resource management problem and focused on some of the important resource management techniques, which include resource allocation, resource provisioning, resource mapping, resource scheduling, and resource adaptation.

Morshedlou and Meybodi [31] proposed a proactive resource-allocation mechanism to reduce the SLA violations. This work also provided the detail control flow of the resource-allocation mechanism. In general, processing a job in clouds can be divided into different phases: Job Submission, Job Placement, Job Execution, Job Migration (if necessary), and Job Complete. Once a job is submitted to the cloud infrastructure, the cloud resource provider has to discover available resources and assign the job to the available resources. This process is known as the initial phase for the job placement. The key mission of this phase is to control the overall resource allocation for each physical server to complete the amount of submitted jobs in an efficient way. Furthermore, when a job is under execution, the job may be transferred from its current assigned resource to a new one. This phase is known as the migration and is usually applied after the job placement phase. In our work, we aim at the initial phase for the job placement and resource-allocation problem rather than the migration phase.

The effect of resource allocation on system performance in terms of static and dynamic control has been examined in studies [12]. Static allocation approaches include the round

robin, optimization, and overbooking, whereas dynamic allocation approaches include reactive and proactive controls. In the round robin approach, virtual machines (VMs) are allocated to one server after another. Therefore, implementation of this mechanism is easy. Each server can have balanced loads when the application demands and resource capacities are homogeneous.

The effectiveness of round robin and the shortest job first (SJF) [13] algorithms for resource allocation in cloud computing has been discussed. In the SJF algorithm, the execution time is assumed to be predefined for each application, which is not practical in cloud computing. Therefore, a modified round robin approach was proposed to improve the resource-allocation efficiency. The resource-allocation problem was addressed using the static server allocation problem (SSAPv) [14]. The overbooking approach can be used to improve the resource utilization. In previous work [15], the authors proposed a novel approach using truncated singular value decomposition to solve the SSAPv problem. These static approaches perform favorably when the number of required VMs and the resource demands are well-defined before allocating the resources.

Some previous works also focused on the task-scheduling problem. Gawali et al. [32] proposed a heuristic approach to improve the turnaround time and response time of tasks in workflow fashion by adopting the modified analytic hierarchy process, bandwidth-aware divisible scheduling, and longest expected processing time preemption approaches. Zhang et al. [33] introduced a two-level asynchronous scheduling model for cloud federation. The proposed model prioritizes the tasks sent by the federation scheduler with two multi-resource fair scheduling algorithms for cloud and federation. In addition, Shukla et al. [34] proposed a mechanism for migrating running streaming dataflow across VMs. Tan et al. [35] proposed a Cooperative Coevolution Genetic Programming (CCGP) hyper-heuristic approach. The proposed approach allocates resources on a two-level architecture that addresses the allocation of containers to VMs and the allocation of VMs to physical machines. Compared to these works, this paper focuses on tackling the essential problem of resource allocation and attempts to reduce the amount of available but unused resources for physical machines.

In another work [16], a reactive control for dynamic resource allocation was proposed. When the loading of a server is higher than a predefined threshold, the reactive control migrates the VM from the server with higher loads to another server with lower loads. Furthermore, if the server load is too low, the reactive approach consolidates the VMs and shuts down the server with low loads. Another approach is to proactively predict the load of servers and allocate resources accordingly rather than waiting for the overloading condition to occur. In previous work [17], the double exponential smoothing method was proposed to predict server loads based on histogram data. Sultan et al. [36] developed an intelligent usage prediction model according to historical resource usage. Authors apply the proposed prediction model to allocate resources dynamically. Tang et al. [37] introduced a dynamical load-balanced scheduling (DLBS) approach to improve the network throughput while balancing workload of data transmissions dynamically. Souravlas [38] addressed the problem of the balanced data flow among data centers. Tantalaki et al. [39] introduced a pipeline-based linear scheduling approach for big data streams. Lattuada et al. [40] presented a resource-allocation approach for big data analytics. The proposed approach adopts a set of run-time optimization-based resource management policies to address the minimum resource requirements with the deadline and to balance the load to avoid the tardiness without enough resources. These works focused on load prediction and data transmissions that are out of the scope in this paper.

Apache Mesos [21] was developed by University of Berkeley Labs and subsequently donated to the Apache Software Foundation. Mesos is a resource managing platform that can be used to not only manage all resources in the distributed system but also allocate resources such as CPU, memory, and storage to applications. In Mesos, a two-level resource-allocation architecture is presented, which uses the dominant resource fairness (DRF) [18] algorithm. The DRF algorithm is a max–min fairness approach for the first phase

of resource scheduling. In DRF, the dominant resources of frameworks are determined, and subsequently, priorities are assigned to frameworks based on the amount of dominant resources. When resources are allocated to frameworks, the scheduler in each framework schedules these resources for executing jobs. However, the heterogeneity of resources disrupts the fairness in DRF. In previous study [19], the dominant resource fairness for heterogeneous (DRFH) was proposed to improve DRF for heterogeneous resource allocation. In DRFH, the share ratio of dominant resources in the server with heterogeneous resources is determined. In the scheduling phase, the number of jobs with dominant resources is calculated. In the allocating phase, DRFH allocates a similar proportion of the dominant resource share ratio among application demands and available resources on each server to reduce the amount of unused resources. Hamzeh et al. [41] proposed a Multi-level Fair Dominant Resource Scheduling (MLF-DRS) algorithm to guarantee the fairness of resource demands based on dominant shares. However, previous approaches were too complicated and are not proposed for the data streaming application in Spark which mainly focuses on the in-memory processing.

Spark utilizes resilient distributed datasets (RDD) [42] to develop a decentralized computational framework. RDD is a data structure for improving Hadoop I/O operations in the MapReduce phase. In RDD, a coarse-grained approach is adopted to record data logs. Two operations occur in RDD, namely transmission and action. Only the action operations could write data into storage for reducing the I/O access time. Two job scheduling algorithms are applied in Spark, namely the FAIR and first-in-first-out algorithms. For the resource-allocation phase, the load-balanced algorithms are used to balance the CPU utilization among cloud servers. When an application is launched, Spark allocates the application to the server with maximum number of available CPU resources. However, the load-balanced algorithm in Spark may result in resource wastage for the available but unused resources when resources are more heterogeneous in clouds.

Allocation of various application demands to heterogeneous resources has received limited attention. The heterogeneity of server capacities affects the resource utilization. For example, when the CPU resources of one Spark server are exhausted, the memory capacity of the same server are still available. In this case, the server cannot afford any application because of insufficient CPU capacity. Consequently, resource wastage leads to low utilization and high execution time in the system. The more heterogeneous the computing resources are, the more amount of available but unused resources influences the system. Therefore, a novel resource-allocation approach was proposed for enhancing the Spark system with the consideration of heterogeneous resources. The proposed algorithm can avoid the amount of available unused resources for each server to enhance the resource utilization and reduce the overall completion time for Spark applications.

3. Minimizing Resource Gap

In this section, details about the proposed MRG approach are described. The concept of resource gap is introduced first and then the MRG algorithm is presented.

3.1. Resource Gap

Assume two servers S_1 and S_2, with two kinds of resources, R_x and R_y. The quantity of R_x and R_y in S_1 is (4,2), and the quantity of R_x and R_y in S_2 is (2,4). Suppose two applications, AP_1 and AP_2, with different resource demands, the resource demand of AP_1 is (2,1) and resource demand of AP_2 is (1,2). Given an algorithm to allocate resources of S_1 to AP_2, the remaining resource of S_1 is (3,0). Thus, in this allocation, three available but unused resources of the R_x are wasted. When the algorithm allocates resources of S_2 to AP_1, the three available but unused resources of R_y are wasted because of the remaining resource of S_2 is (0,3). After the previous resource allocation, the system cannot afford more resource requests of other applications even though the overall system remains available R_x and R_y resources as (3,3). Accordingly, the higher heterogeneous the resource demand

is, the more resource gap will be. As a result, the more available but unused resources are wasted in the system.

To solve the resource wastage problem for the available but unused resources, the MRG algorithm was proposed to calculate the degree of resource wastage and improve resource utilization. The difference between the quantities of exhausted resources and remaining resources is termed as the *resource gap*. When a resource is exhausted and the quantity of remaining resources is the smallest, resource wastage for the available but unused resources is reduced. Figure 1 depicts an example to illustrate the resource gap. Assume three resources R_1, R_2, and R_3 in the system. The initial quantities of R_2 and R_3 were considerably higher than that of R_1. That is, before resource allocation, the original resource gap was two between R_1 and R_2 and four between R_1 and R_3. After resource allocation, the resource gap among remaining resources was one between R_1 and R_2 and two between R_1 and R_3. Accordingly, the resource gap between heterogeneous resources could be minimized after an efficient resource-allocation policy. Thus, all three resources are well utilized to prevent resource wastage.

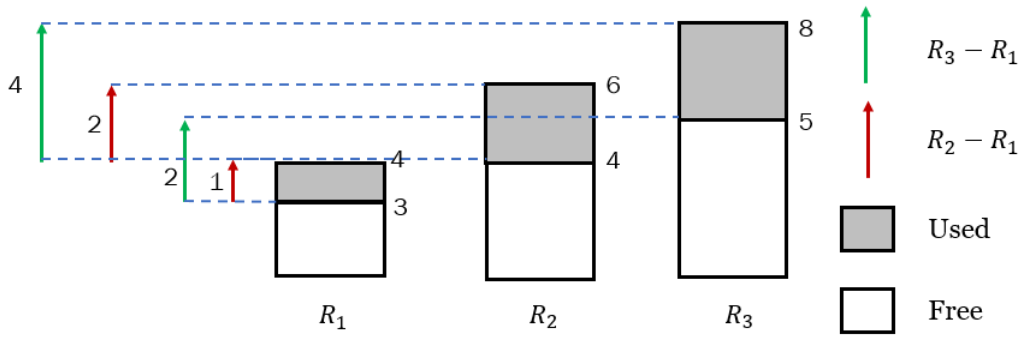

Figure 1. Differences between quantities of exhausted resources and remaining resources.

3.2. MRG Algorithm

To minimize the resource gap among the heterogeneity of resources, the MRG mechanism was proposed in this study. Initially, the possible resource wastage is calculated in the MRG algorithm when an application is launched. Then, the server with the smallest resource gap is determined to allocate resources to this application. Therefore, each server has a low resource wastage for the available but unused resources and the system utilization is enhanced. In this study, the set S of servers was $\{S_1, S_2, \ldots, S_s\}$, and the set R of resources was $\{R_1, R_2, \ldots, R_r\}$. Available resources of the server K ($K \in S$) are $A_{KR} = \{A_{KR_1}, A_{KR_2}, \ldots, A_{KR_r}\}$. The application demand for resources is presented as the vector $D = \{D_{R_1}, D_{R_2}, \ldots, D_{R_r}\}$.

The MRG algorithm is invoked when the application demand is submitted for allocating available resources. First, the following is calculated in MGR:

$$A' = \{A_{KR_i} - D_{R_i}\}, \qquad \forall i \in R \tag{1}$$

This parameter is determined for calculating the remaining amount of resources if the algorithm allocates the candidate server K to the application. Then, the smallest A' with A_{KR_m} is determined, which indicates that the resource R_m in server K is going to be depleted. All resource gaps in server K are then calculated in the MRG algorithm for determining the accumulated resource gap, denoted by ARG_K, as follows:

$$\sum_{i=R_1,\ i\neq R_m}^{R_r} (A'_{KR_i} - A'_{KR_m}), \quad \forall K \in S \tag{2}$$

Finally, the target server T_S with the smallest ARG is determined and this server is allocated to the application. The pseudo codes for the proposed MRG algorithm are shown in Figure 2.

Minimize Resource Gap (MRG) Algorithm

Input: $D_R(D_{R_1}, D_{R_2,} \dots, D_{R_r}), S(S_1, S_2, \dots, S_s)$

Output: Ts

```
main MRG()
begin
    for S_i ∈ [S_1, S_2, ..., S_x]
        A_SR = available resources of S_i
        D_R = resource demands of the application
        ARG = CalculateServerARG(A_SR, D_R) by Equation (2)
        if ARG < minARG then
            replace minARG
            update Ts
        end if
    end for
    return Ts
end

CalculateServerARG(A_SR(A_SR1, A_SR2, ..., A_SRr), D_R(D_R1, D_R2, ..., D_Rr))
begin
    minAKR = index = 0
    for R_j ∈ [R_1, R_2, ..., R_r] by Equation (1)
        A_SRj = A_SRj − D_Rj
        if A_SRj < minAKR then
            minAKR = A_SRj
            index = R_j
        end if
    end for
    ARG = 0
    for R_k ∈ [R_1, R_2, ..., R_r], R_k != index
        ARG = ARG + (A_SRk − D_Rk)
    end for
    return ARG
end
```

Figure 2. Pseudo codes for the proposed minimizing resource gap (MRG) algorithm.

The following examples S_1, S_2, AP_1, AP_2 were applied to demonstrate the effectiveness of the proposed algorithm. When only two resources are available, Equation (2) is simplified to the following expression:

$$\text{Min}\left|\left(A_{iR_1} - D_{R_1}\right) - \left(A_{iR_2} - D_{R_2}\right)\right|, \quad i \in (S_1, S_2) \tag{3}$$

Tables 1 and 2 list the examples to illustrate allocation steps when allocating resources for AP_1 and AP_2. When the resource is allocated to AP_1, the ARG of S_1 is $|(4-2)-(2-1)| = 1$ and the ARG of S_2 is $|(2-2)-(4-1)| = 3$. Because S_1 has a smaller ARG, the MRG algorithm allocates AP_1 to be on S_1. When allocating resources for AP_2, the ARG of S_1 is $|(4-1)-(2-2)| = 3$, whereas the ARG of S_2 is $|(2-1)-(4-2)| = 1$. Because S_2 has a smaller ARG, AP_2 is allocated to S_2. These examples show that the proposed algorithm can allocate resources effectively to prevent a high resource gap.

Table 1. Allocation steps for AP_1 using the MRG algorithm.

Allocation Steps	Server S_1		Server S_2		Allocated
	A_{S_1R}	ARG_{S_1}	A_{S_2R}	ARG_{S_2}	
AP_1 D_R = (2,1)	(4,2)	$\|(4-2)-(2-1)\| = 1$	(2,4)	$\|(2-2)-(4-1)\| = 3$	Server S_1

Gary means the decision in the case.

Table 2. Allocation steps for AP_2 using MRG algorithm

Allocation Steps	Server S_1		Server S_2		Allocated
	A_{S_1R}	ARG_{S_1}	A_{S_2R}	ARG_{S_2}	
AP_2 D_R = (1,2)	(4,2)	$\|(4-1)-(2-2)\| = 3$	(2,4)	$\|(2-1)-(4-2)\| = 1$	Server S_2

Gary means the decision in the case.

To further demonstrate that MRG can reduce resource wastage for the available but unused resources, the proposed algorithm was compared with the default load-balanced algorithm in Apache Spark. In the default algorithm, the servers are allocated with the maximal free CPU cores to the application. Assume that S_1 has resources R_1, R_2, and R_3 with values of (4, 6, 8) and S_2 has resources of R_1, R_2, and R_3 with values of (8, 6, 4), respectively. The resource demands of AP_1 is (1, 2, 3) and that of AP_2 is (4, 3, 2). Suppose that these two applications are allocated one after another. Therefore, two cases are possible for the resource-allocation policy. The first case starts with the allocation step for AP_2 and the other case starts with the allocation step for AP_1. Tables 3 and 4 illustrate allocation steps for Case 1, applying the default algorithm in Apache Spark and the proposed MRG algorithm, respectively. Tables 5 and 6 illustrate allocation steps for Case 2.

Table 3. Allocation steps for Case 1 using load-balanced algorithm.

Allocation Steps	Server S_1		Server S_2		Allocated
	A_{S_1R}	Max Free CPU Cores	A_{S_2R}	Max Free CPU Cores	
AP_2 D_R= (4,3,2)	(4,6,8)		(8,6,4)	(8,6,4) − (4,3,2) = (4,3,2)	Server S_2
AP_1 D_R= (1,2,3)	(4,6,8)	(4,6,8) − (1,2,3) = (3,4,5)	(4,3,2)		Server S_1
AP_2 D_R= (4,3,2)	(3,4,5)		(4,3,2)	(4,3,2) − (4,3,2) = (0,0,0)	Server S_2
AP_1 D_R= (1,2,3)	(3,4,5)	(3,4,5) − (1,2,3) = (2,2,2)	(0,0,0)		Server S_1

Gary means the decision in the case.

Table 4. Allocation steps for Case 1 using the MRG algorithm.

Allocation Steps	Server S_1		Server S_2		Allocated
	A_{S_1R}	ARG_{S_1}	A_{S_2R}	ARG_{S_2}	
AP_2 D_R= (4,3,2)	(4,6,8)	(4,6,8) − (4,3,2) = (0,3,6) (3-0) + (6-0) = 9	(8,6,4)	(8,6,4) − (4,3,2) = (4,3,2) (4-2) + (3-2) = 3	Server S_2
AP_1 D_R= (1,2,3)	(4,6,8)	(4,6,8) − (1,2,3) = (3,4,5) (4-3) + (5-3) = 3	(4,3,2)		Server S_1
AP_2 D_R= (4,3,2)	(3,4,5)		(4,3,2)	(4,3,2) − (4,3,2) = (0,0,0) (0-0) + (0-0) = 0	Server S_2
AP_1 D_R= (1,2,3)	(3,4,5)	(3,4,5) − (1,2,3) = (2,2,2) (2-2) + (2-2) = 0	(0,0,0)		Server S_1

Gary means the decision in the case.

Table 5. Allocation steps for Case 2 using the load-balanced algorithm.

Allocation Steps	Server S_1		Server S_2		Allocated
	A_{S_1R}	Max Free CPU Cores	A_{S_2R}	Max Free CPU Cores	
AP_1 D_R= (1,2,3)	(4,6,8)		(8,6,4)	(8,6,4) − (1,2,3) = (7,4,1)	Server S_2
AP_2 D_R= (4,3,2)	(4,6,8)	(4,6,8) − (4,3,2) = (0,3,6)	(7,4,1)		Server S_1
AP_1 D_R= (1,2,3)	(0,3,6)		(7,4,1)		
AP_2 D_R= (4,3,2)	(0,3,6)		(7,4,1)		

Gary means the decision in the case.

Table 6. Allocation steps for Case 2 using the MRG algorithm.

Allocation Steps	Server S_1		Server S_2		Allocated
	A_{S_1R}	ARG_{S_1}	A_{S_2R}	ARG_{S_2}	
AP_1 D_R= (1,2,3)	(4,6,8)	(4,6,8) − (1,2,3) = (3,4,5) (4-3) + (5-3) = 3	(8,6,4)	(8,6,4) − (1,2,3) = (7,4,1) (4-1) + (7-1) = 9	Server S_1
AP_2 D_R= (4,3,2)	(3,4,5)		(8,6,4)	(8,6,4) − (4,3,2) = (4,3,2) (3-2) + (4-2) = 3	Server S_2
AP_1 D_R= (1,2,3)	(3,4,5)	(3,4,5) − (1,2,3) = (2,2,2) (2-2) + (2-2) = 0	(4,3,2)		Server S_1
AP_2 D_R= (4,3,2)	(2,2,2)		(4,3,2)	(4,3,2) − (4,3,2) = (0,0,0) (0-0) + (0-0) = 0	Server S_2

Gary means the decision in the case.

In Case 1, the load-balanced algorithm and the proposed MRG algorithm could allocate resources to satisfy the application demands for AP_1 and AP_2. However, in Case 2, the load-balanced algorithm can only satisfy two application demands for one AP_1 and one AP_2. Resource wastage for the available but unused resources is higher for the load-balanced algorithm in this case. The proposed MRG algorithm can satisfy all the resource demands for AP_1 and AP_2. This is because MRG minimizes the resource gap for reducing the amount of available but unused resources.

The aforementioned cases indicate that the MRG algorithm can not only reduce the resource gap in the heterogeneous resource environment but also improve resource utilization. The load-balanced algorithm performs satisfactorily in the environment with homogeneous resources but results in poor resource utilization in the environment with heterogeneous resources. The time complexity of our MRG algorithm depends on the factors of S servers and R kinds of resource capacities. For each server S, the proposed algorithm calculates the ARG by Equation (2). for determining the accumulated resource gap. Then, the proposed algorithm finds out the target server Ts with the smallest ARG and allocate the resource of this server to the application. So, the most time consuming of the proposed algorithm is to find out the Ts. Therefore, the time complexity of the proposed algorithm is $O(S \times R)$.

4. Experimental Results

In this section, the performance of the proposed approaches is evaluated. We first describe the environmental settings and methodology of the experiment, and subsequently, the experimental results of the proposed approach.

4.1. Experimental Environment

MRG was implemented in Apache Spark, which is an open-source distributed computing framework with the master–slave architecture. Docker [43] was used to execute Apache Spark on three IBM Blade Center HS22 servers. The native orchestration of Docker Swarm [44] was used for the management of container clusters across servers. The master node on one server was launched using Docker Swarm and a container running as the worker node was launched on other two servers. To provide the heterogeneity of runtime environment, one container had 16 CPUs with 8 GB RAM, and the other had 8 CPUs and 16 GB RAM.

Two applications were used in the study, Spark Pi and Spark PageRank. These applications have distinct benchmarks. Spark Pi is a classic MapReduce framework and a CPU-intensive computing application for big data analysis in Apache Spark. More CPU resources are required to improve the computing efficiency of Spark Pi. Spark PageRank is a typical page ranking algorithm in which a relative score of the importance of website pages is evaluated. To cache a large set of intermediate data in memory, Spark PageRank is memory intensive and the most critical effect on performance is memory utilization. To measure system performance, the overall completion time was evaluated under various combinations of application resource demands and server capacities.

4.2. Experimental Results

Experimental results are presented for the proposed MRG approach. In the following experiments, the resource demand for high CPU cores resulted in the execution of the Spark Pi application and that of a high memory capacity resulted in the Spark PageRank application.

4.2.1. Different Ratios of Application Resource Demands

To evaluate the performance on different application resource demands, three ratios for resource demands, ([1:2], [1:4], and [1:6]) were assumed in the first experiment. We used these different ratios in experiments to evaluate the performance of heterogeneous degree. The higher the ratio we applied, the more heterogeneous the resource demands will be. The ratio of [1:2] represented that one application demand requires a resource with 1 CPU core and 2 GB memory for running Spark PageRank, and another application demand requires 2 CPU cores and 1 GB memory for running Spark Pi. Furthermore, [1:4] and [1:6] represented that the ratio of CPU cores and memory capacity is ((1,4) and (4,1)), and ((1,6) and (6,1)), respectively, for two different applications. Both Spark Pi and Spark PageRank applications were used in the Spark system and different resource demands from these two kinds of applications were submitted to the system as processing jobs. The number of submitted jobs for each application was increased from 4, 8, 16 to 32 for each round of the ratio.

The experimental results are depicted in Figures 3–5. These results illustrated that, when the resource demand of an application was closer to the available resources of the server, resource utilization was higher and performance improvement was obvious. The proposed MRG algorithm improved the average performance with 6.5%, 8.5%, and 1.9% for each ratio. Particularly, the performance gain was up to 12.4% in the ratio of [1:2]. The performance gain did not increase with the number of submitted jobs because all application demands exceeded the total system resources. In addition, when the resource demands of an application do not match the available resource of a server, MRG does not waste any unused resource, unlike the load-balanced algorithm.

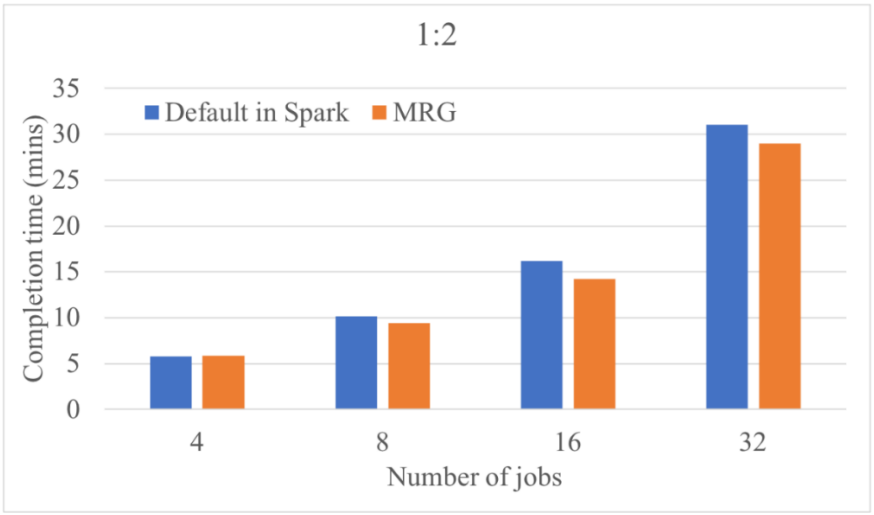

Figure 3. Ratio of resource demand is [1:2] in which one application requires 1 CPU core and 2 GB RAM and another application requires 2 CPU cores and 1 GB memory.

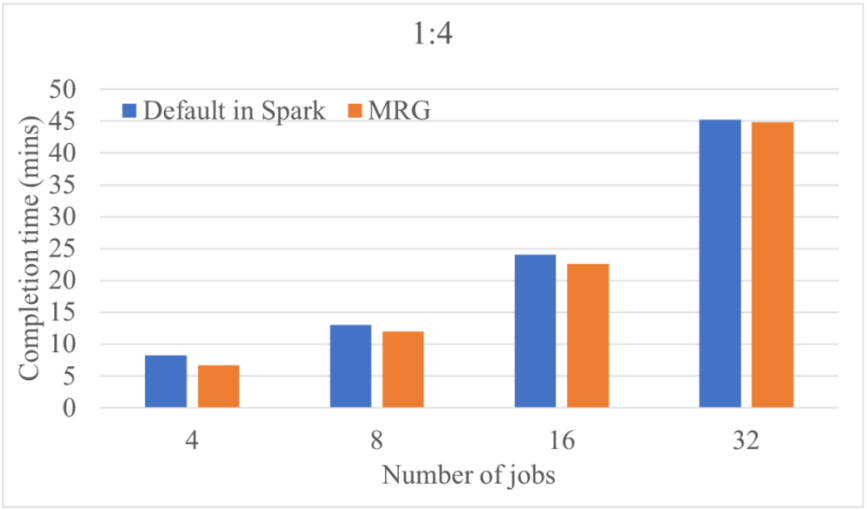

Figure 4. Ratio of resource demand is [1:4] in which one application requires 1 CPU core and 4 GB RAM and another application requires 4 CPU cores and 1 GB memory.

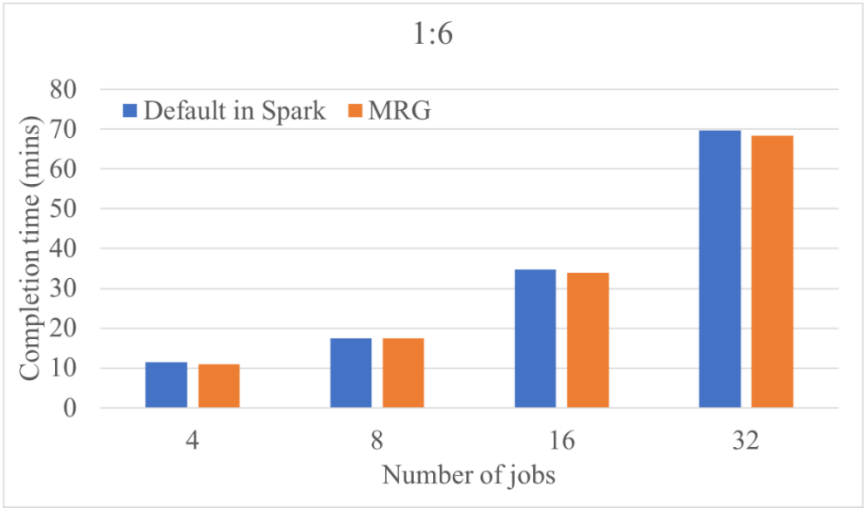

Figure 5. Ratio of resource demand is [1:6] in which one application requires 1 CPU core and 6 GB RAM and another application requires 6 CPU cores and 1 GB memory.

Overall, the results of this experiment proved that system performance improved with MRG compared with the load-balanced algorithm. The more jobs submitted to the system will increase the waiting time for available resources for processing the application jobs. In addition, a resource gap occurred while some resource capacities were exhausted and other resource capacities on the same server were still available. Furthermore, higher resource utilization was observed in the MRG algorithm so as to reduce the completion time. As a result, improving the resource utilization by the MRG algorithm can reduce the overall completion time.

4.2.2. Mixed Ratios of Application Resource Demands

The performance of the MRG algorithm was evaluated when the resource demands of applications were more complex. A mixture of cases with the resource demands in [1:2], [1:4], and [1:6] was evaluated. Each set of the experiment consisted of six applications with the resource demand vector of CPU and memory by (1,2), (2,1), (1,4), (4,1), (1,6), and (6,1). In this experiment, the number of sets was increased from one to eight for each round of evaluation.

Experimental results are depicted in Figure 6. As shown in Figure 6, the experimental result showed that the performance of MRG increased with an increase in resource demands. When the number of sets was two, both the load-balanced algorithm and the MRG algorithm could satisfy the low resource demand of applications. However, when the number of applications increased, the proposed MRG algorithm exhibited a lower completion time than the load-balanced algorithm. The results showed that the MRG approach outperform the load-balanced algorithm by up to 24.7% in terms of the completion time. This is because the degree of resource gap was higher for the load-balanced algorithm. By contrast, the MRG algorithm could prevent the resource gap and improve the resource utilization to reduce the overall completion time.

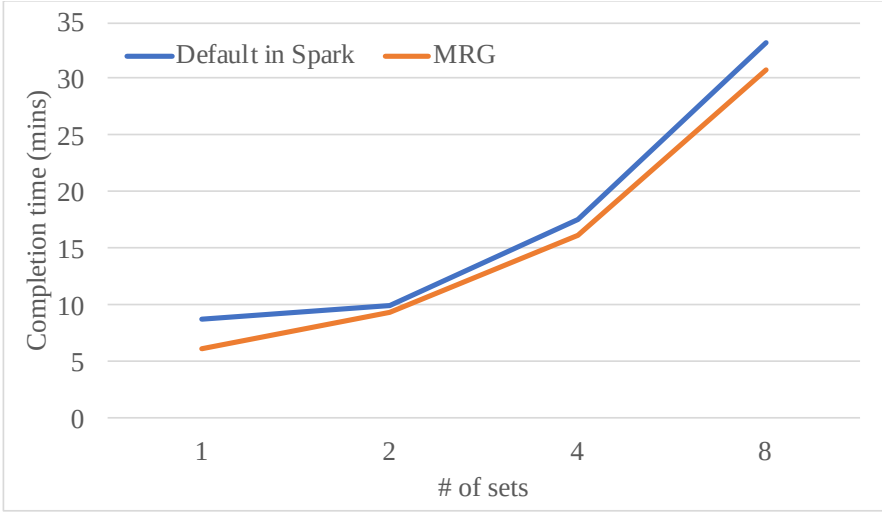

Figure 6. Performance comparisons of mixed ratios of application demands.

4.2.3. Various Ratios of Server Capacities

MRG performance under various combinations of server capacities with a fixed ratio of resource demands was evaluated. Two servers were used with various ratios of CPU cores and memory capacities in the range from (8,2), (8,4), (8,8), (8,16) to (8,32). Two applications, Spark Pi and Spark PageRank, were used in the system with the ratio of [1:2]. Other ratios were not conducted in this experiment because of the limitation of server capacities. When the ratio of resource demands is smaller, there are more jobs submitted to the experimental system. Therefore, the lower ratio of resource demands can be experimented with more performance results to evaluate the resource provisioning. One application required 1 CPU core and 2 GB memory to execute Spark PageRank, and the other application required 2 CPU cores and 1 GB memory for running Spark Pi. Each round of the experiment had nine applications submitted to the system with varied server capacities.

In general, the higher resource capacities the server has, the more performance gain the MRG approach will be for the corresponding application. Figure 7 illustrates that the overall execution time can be reduced when one of the server resource capacities increases from (8,2) to (8,4). The results showed that the performance improvement was not highly relevant to the increasing memory capacities, which is because Spark Pi is CPU-intensive. Increasing memory capacities does not considerably reduce the execution time. On the contrary, Figure 8 depicted performance improvement when running the memory-intensive application Spark PageRank with various sizes of memory capacities. Results revealed that, when one of the server capacities was (8,2), the completion time could be improved as the other server capacities increased from (8,2) to (8,16). Accordingly, the higher capacities the memory has, the more performance gain the MRG approach will have.

Finally, in this study, MRG performance was evaluated after submitting nine Spark Pi and nine Spark PageRank applications to the system. As depicted in Figure 9, the proposed MRG approach could considerably reduce the overall execution time when the resource capacities were increasing. This is because the server could allocate resources for both CPU- and memory-intensive applications with sufficient resource capacities and efficient MRG allocation method. The performance gain was achieved more than 60% when the ratios of resource capacities were over (8,8). In particular, the improved performance can be up to 64.7% in the case of (8,8) and (8,16) for running both CPU- and memory-intensive applications.

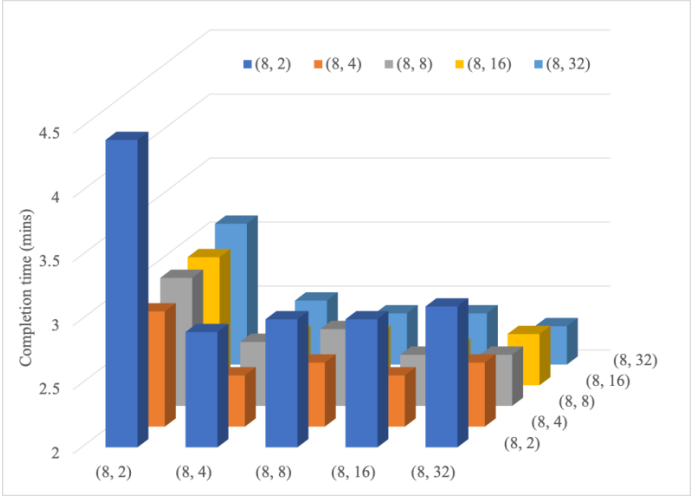

Figure 7. Performance evaluations of various server capacities to run Spark Pi.

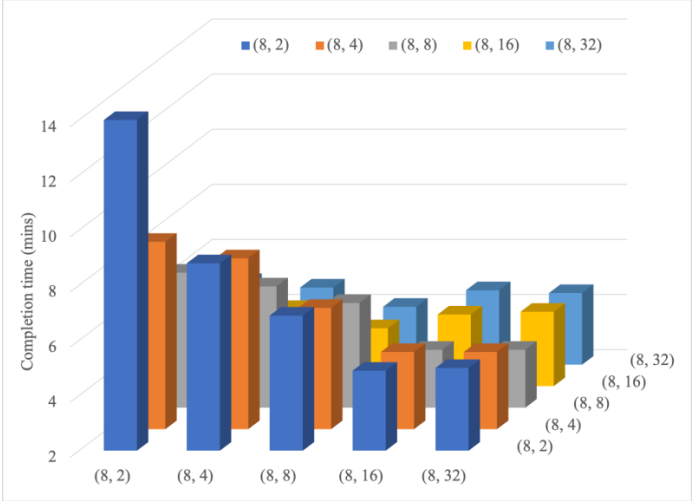

Figure 8. Performance evaluations of various server capacities to run Spark PageRank.

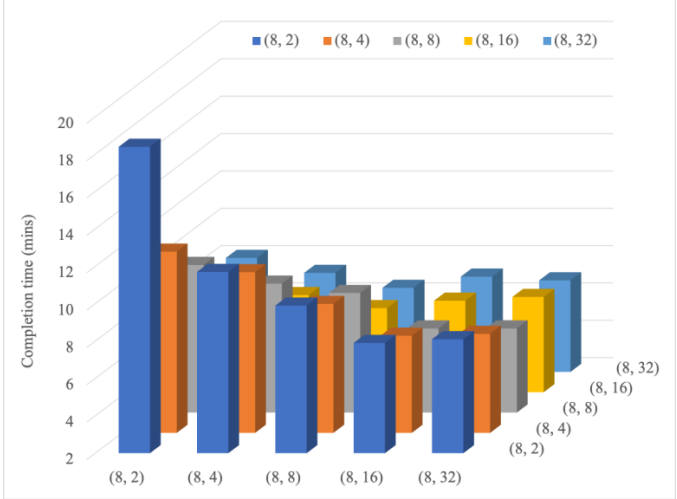

Figure 9. Performance evaluations of various server capacities to run both Spark Pi and Spark PageRank.

5. Conclusions and Future Work

On-demand and dynamic resource provisioning plays a crucial role in improving the resource utilization in cloud computing. The efficiency of the resource-allocation mechanism affects system performance, particularly in the system with heterogeneous resources. When designing effective resource allocation for heterogeneous resources, the heterogeneity of resource capacities should be considered to avoid resource wastage. Resource gap is a phenomenon in which some resource capacities are exhausted, whereas other resources on the same server are still available. The more heterogeneous the computing resources are, the more influence the resource gap has on the system. A computing server may not satisfy all the demands of resource capacities from an application. Therefore, the system exhibits low resource utilization and resource wastage for the available but unused resources.

In this study, a novel resource-allocation approach, MRG, was proposed to solve the resource wastage problem. In MRG, resource demands among different applications, for example, CPU-intensive and memory-intensive applications, are considered for enhancing resource usage. When applications are submitted to the system, MRG calculates the possible resource wastage and determines the server with the smallest resource gap to allocate resources to each application. Therefore, each server can avoid the amount of available but unused resources to enhance system utilization by up to 24.7% in terms of the overall completion time.

MRG was implemented in Apache Spark to demonstrate MRG performance. Two applications with different resource demands (Spark Pi and Spark PageRank) were applied. The performance metric of overall complete time was measured under the various combination sets of different application resource demands and server capacities. Experimental results indicated the superiority of the proposed MRG approach over the load-balanced approach in Apache Spark. Furthermore, MRG can considerably reduce the overall execution time as more servers can allocate resources for both CPU- and memory-intensive applications. The performance gain can be achieved by up to 64.7% when applying the MRG approach.

In the future, we will extend the MRG approach for improving system utilization with more dimensions of resource heterogeneities. Resource allocation is also one of the important issues in mobile cloud computing [29,30]. After the offloaded task moving from the mobile device into the cloud, the cloud provider has to allocate enough resources to this task. In this case, task migration could be a further finetuned mechanism to enhance the job processing. Applying machine learning to enhance the MRG algorithm is vital as well for future studies.

Author Contributions: Conceptualization, T.-L.W., K.-C.L., and W.-C.C.; data curation, Y.-H.L. and K.-C.H.; investigation, W.-C.C.; methodology, T.-L.W. and K.-C.L.; resources, H.-C.H.; validation, T.-L.W.; formal analysis, W.-C.C.; writing—original draft preparation, W.-C.C.; writing—review & editing, W.-C.C. and K.-C.L. All authors have read and agreed to the published version of the manuscript.

Funding: This research received no external funding.

Acknowledgments: This study was sponsored by the Ministry of Science and Technology, Taiwan, under contract numbers: MOST 107-2221-E-142-004-MY3 and MOST 107-2218-E-006-055, and by the "Intelligent Manufacturing Research Center" (iMRC) from the Featured Areas Research Center Program within the framework of the Higher Education Sprout Project by the Ministry of Education, Taiwan.

Conflicts of Interest: The authors declare no conflict of interest.

References

1. Gartner Says 5.8 Billion Enterprise and Automotive IoT Endpoints Will Be in Use in 2020. Available online: https://www.gartner.com/en/newsroom/press-releases/2019-08-29-gartner-says-5-8-billion-enterprise-and-automotive-io (accessed on 1 October 2020).
2. Apache Hadoop. Available online: http://hadoop.apache.org (accessed on 1 October 2020).

3. Apache Spark. Available online: https://spark.apache.org/ (accessed on 1 October 2020).
4. Dean, J.; Ghemawat, S. Mapreduce: Simplified data processing on large clusters. In Proceedings of the 6th USENIX Symposium on Operating Systems Design and Implementation, San Francisco, CA, USA, 6–8 December 2004; pp. 137–150.
5. Taran, V.; Alienin, O.; Stirenko, S.; Gordienko, Y.; Rojbi, A. Performance evaluation of distributed computing environments with Hadoop and Spark frameworks. In Proceedings of the 2017 IEEE International Young Scientists Forum on Applied Physics and Engineering, Lviv, Ukraine, 17–20 October 2017; pp. 80–83.
6. Samadi, Y.; Zbakh, M.; Tadonki, C. Performance comparison between Hadoop and Spark frameworks using HiBench benchmarks. *Concurr. Comput. Pract. Exp.* **2018**, *30*, e4367. [CrossRef]
7. Yousafzai, A.; Gani, A.; Noor, R.M.; Sookhak, M.; Talebian, H.; Shiraz, M.; Khan, M.K. Cloud resource allocation schemes: Review, taxonomy, and opportunities. *Knowl. Inf. Syst.* **2017**, *50*, 347–381. [CrossRef]
8. Manvi, S.S.; Shyam, G.K. Resource management for Infrastructure as a Service (IaaS) in cloud computing: A survey. *J. Network Comput. Appl.* **2014**, *41*, 424–440. [CrossRef]
9. Kumar, M.; Sharma, S.C.; Goel, A.; Singh, S.P. A comprehensive survey for scheduling techniques in cloud computing. *J. Netw. Comput. Appl.* **2019**, *143*, 1–33. [CrossRef]
10. Strumberger, I.; Bacanin, N.; Tuba, M.; Tuba, E. Resource Scheduling in Cloud Computing Based on a Hybridized Whale Optimization Algorithm. *Appl. Sci.* **2019**, *9*, 4893. [CrossRef]
11. Armbrust, M.; Fox, A.; Griffith, R.; Joseph, A.D.; Katz, R.; Konwinski, A.; Lee, G.; Patterson, D.; Rabkin, A.; Stoica, I.; et al. A View of Cloud Computing. *Commun. ACM* **2010**, *53*, 50–58. [CrossRef]
12. Wolke, A.; Bichler, M.; Setzer, T. Planning vs. dynamic control: Resource allocation in corporate clouds. *IEEE Trans. Cloud Comput.* **2016**, *4*, 322–335. [CrossRef]
13. Pradhan, P.; Behera, P.K.; Ray, B.N.B. Modified round robin algorithm for resource allocation in cloud computing. *Procedia Comput. Sci.* **2016**, *85*, 878–890. [CrossRef]
14. Speitkamp, B.; Bichler, M. A mathematical programming approach for server consolidation problems in virtualized data centers. *IEEE Trans. Serv. Comput.* **2010**, *3*, 266–278. [CrossRef]
15. Setzer, T.; Bichler, M. Using matrix approximation for high dimensional discrete optimization problems: Server consolidation based on cyclic time-series data. *Eur. J. Oper. Res.* **2013**, *227*, 62–75. [CrossRef]
16. Wood, T.; Shenoy, P.; Venkataramani, A.; Yousif, M. Sandpiper: Black-box and gray-box resource management for virtual machines. *J. Netw. Comput. Appl.* **2009**, *53*, 2923–2938. [CrossRef]
17. Hamilton, J.D. *Time Series Analysis*; Princeton University Press: Princeton, NJ, USA, 1994.
18. Ghodsi, A.; Zaharia, M.; Hindman, B.; Konwinski, A.; Shenker, S.; Stoica, I. Dominant resource fairness: Fair allocation of multiple resource types. In Proceedings of the 8th USENIX Conference on Networked Systems Design and Implementation, Boston, MA, USA, 30 March–1 April 2011; pp. 323–336.
19. Wang, W.; Liang, B.; Li, B. Multi-resource fair allocation in heterogeneous cloud computing systems. *IEEE Trans. Parallel Distrib. Syst.* **2015**, *26*, 2822–2835. [CrossRef]
20. Vavilapalli, V.K.; Murthy, A.C.; Douglas, C.; Agarwal, S.; Konar, M.; Evans, R.; Graves, T.; Lowe, J.; Shah, H.; Seth, S.; et al. Apache Hadoop YARN: Yet another resource negotiator. In Proceedings of the 4th annual Symposium on Cloud Computing, Santa Clara, CA, USA, 1–3 October 2013; pp. 1–16.
21. Hindman, B.; Konwinski, A.; Zaharia, M.; Ghodsi, A.; Joseph, A.D.; Katz, R.H.; Shenker, S.; Stoica, I. *Mesos: A Platform for Fine-Grained Resource Sharing in the Data Center*; Technical Report No. UCB/EECS-2010-87; University of California Berkeley: Berkeley, CA, USA, 2010.
22. Wei, L.; Foh, C.H.; He, B.; Cai, J. Towards Efficient Resource Allocation for Heterogeneous Workloads in IaaS Clouds. *IEEE Trans. Cloud Comput.* **2018**, *6*, 264–275. [CrossRef]
23. Bierzynski, K.; Escobar, A.; Eberl, M. Cloud, fog and edge: Cooperation for the future? In Proceedings of the Second International Conference on Fog and Mobile Edge Computing, Valencia, Spain, 8–11 May 2017; pp. 62–67.
24. Sajjad, H.P.; Danniswara, K.; Al-Shishtawy, A.; Vlassov, V. SpanEdge: Towards unifying stream processing over central and near-the-edge data centers. In Proceedings of the IEEE/ACM Symposium on Edge Computing, Washington, DC, USA, 27–28 October 2016; pp. 168–178.
25. Varshney, P.; Simmhan, Y. Demystifying fog computing: Characterizing architectures, applications and abstractions. In Proceedings of the 1st International Conference on Fog and Edge Computing, Madrid, Spain, 14–15 May 2017; pp. 115–124.
26. Jain, R.; Tata, S. Cloud to edge: Distributed deployment of process-aware IoT applications. In Proceedings of the 1st International Conference on Edge Computing, Honolulu, HI, USA, 25–30 June 2017; pp. 182–189.
27. Zaharia, M.; Chowdhury, M.; Franklin, M.J.; Shenker, S.; Stoica, I. Spark: Cluster computing with working sets. In Proceedings of the 2nd USENIX Conference on Hot Topics in Cloud Computing, Boston, MA, USA, 22 June 2010; p. 10.
28. Powered by Spark. Available online: https://spark.apache.org/powered-by.html (accessed on 1 October 2020).
29. Shakarami, A.; Ghobaei-Arani, M.; Masdari, M.; Hosseinzadeh, M. A Survey on the Computation Offloading Approaches in Mobile Edge/Cloud Computing Environment: A Stochastic-based Perspective. *J. Grid Comput.* **2020**. [CrossRef]
30. Arun, C.; Prabu, K. Resource Allocation in Mobile Cloud Computing: A Survey. *Int. Res. J. Eng. Technol.* **2018**. Available online: https://www.irjet.net/archives/V5/i7/IRJET-V5I7162.pdf (accessed on 28 October 2020).

31. Morshedlou, H.; Meybodi, M.R. Decreasing Impact of SLA Violations: A Proactive Resource Allocation Approach for Cloud Computing Environments. *IEEE Trans. Cloud Comput.* **2014**, *2*, 156–167. [CrossRef]
32. Gawali, M.B.; Shinde, S.K. Task scheduling and resource allocation in cloud computing using a heuristic approach. *J. Cloud Comput. Adv. Syst. Appl.* **2018**, 7. [CrossRef]
33. Zhang, G.; Lu, R.; Wu, W. Multi-resource fair allocation for cloud federation. In Proceedings of the 2019 IEEE International Conference on High Performance Computing and Communications, Zhangjiajie, China, 10–12 August 2019.
34. Shukla, A.; Simmhan, Y. Toward reliable and rapid elasticity for streaming dataflows on clouds. In Proceedings of the 2018 IEEE 38th International Conference on Distributed Computing Systems, Vienna, Austria, 2–6 July 2018.
35. Tan, B.; Ma, H.; Mei, Y.; Zhang, M. A Cooperative Coevolution Genetic Programming Hyper-Heuristic Approach for On-line Resource Allocation in Container-based Clouds. *IEEE Trans. Cloud Comput.* **2020**. [CrossRef]
36. Sultan, S.; Asad, A.; Abubakar, M.; Khalid, S.; Ahmed, S.; Wali, A. Dynamic cloud resources allocation. In Proceedings of the 8th International Conference on Information and Communication Technologies, Karachi, Pakistan, Pakistan, 16–17 November 2019.
37. Tang, F.; Yang, L.T.; Tang, C.; Li, J.; Guo, M. A Dynamical and Load-Balanced Flow Scheduling Approach for Big Data Centers in Clouds. *IEEE Trans. Cloud Comput.* **2016**. [CrossRef]
38. Souravlas, S. ProMo: A Probabilistic Model for Dynamic Load-Balanced Scheduling of Data Flows in Cloud Systems. *Electronics* **2019**, *8*, 990. [CrossRef]
39. Tantalaki, N.; Souravlas, S.; Roumeliotis, M.; Katsavounis, S. Pipeline-Based Linear Scheduling of Big Data Streams in the Cloud. *IEEE Access* **2020**, *8*, 117182–117202. [CrossRef]
40. Lattuada, M.; Barbierato, E.; Gianniti, E.; Ardagna, D. Optimal Resource Allocation of Cloud-Based Spark Applications. *IEEE Trans. Cloud Comput.* **2020**. [CrossRef]
41. Hamzeh, H.; Meacham, S.; Virginas, B.; Khan, K.; Phalp, K. MLF-DRS: A Multi-level fair resource allocation algorithm in heterogeneous cloud computing systems. In Proceedings of the 2019 IEEE 4th International Conference on Computer and Communication Systems, Singapore, 23–25 February 2019.
42. Zaharia, M.; Chowdhury, M.; Das, T.; Dave, A.; Ma, J.; McCauley, M.; Franklin, M.; Shenker, S.; Stoica, I. *Resilient Distributed Datasets: A Fault-Tolerant Abstraction for In-Memory Cluster Computing*; Technical Report No. UCB/EECS-2011-82; EECS Department, University of California Berkeley: Berkeley, CA, USA, 2011.
43. Docker. Available online: https://www.docker.com/ (accessed on 1 October 2020).
44. Docker Swarm. Available online: https://docs.docker.com/engine/swarm/ (accessed on 1 October 2020).