The Journal of Supercomputing (2020) 76:1931–1958
https://doi.org/10.1007/s11227-018-2505-4

Fine-grained scheduling in multi-resource clusters

Mosong Zhou[1] · Xiaoshe Dong[1] · Heng Chen[1] · Xingjun Zhang[1]

Published online: 17 August 2018
© Springer Science+Business Media, LLC, part of Springer Nature 2018

Abstract
In multi-resource clusters, many schedulers allocate resources based on fixed quantities. However, fixed allocations can easily lead to resource fragmentation and over-commitment problems, which may result in lower resource utilization and performance degradation. This paper proposes a fine-grained method (FGM) to improve the allocation granularity of resource allocation. This method divides tasks into execution stages according to the task requirement estimated using similar tasks at the runtime. Then, task resource requirements are matched with the available server resources by stages to refine two aspects of allocation granularity: allocation duration and allocation quantity. In addition, the FGM may over-allocate resources deliberately to further improve resource utilization and performance. The paper tested the FGM in three environments using both online and offline workloads. The test results show that the FGM can resolve resource fragmentation and over-commitment problems by significantly improving resource utilization and performance with acceptable fairness and scheduling response times.

Keywords Allocation granularity · Scheduling · Resource management · Cluster · Cloud computing

1 Introduction

Resource management and job scheduling are important tasks in a cluster computing platform. The effectiveness of traditional resource management methods decreases as the diversity and dynamicity of workloads increase [1] because of following reasons. On the one hand, the resource allocation is larger than the resource requirement of task if resource allocation granularity is coarse. And in this case, the surplus resource in allocation cannot be allocated to other tasks and the resource is wasted in the form of fragmentation. In the other hand, the resource may be allocated to too many tasks when

✉ Xingjun Zhang
xjzhang@mail.xjtu.edu.cn

1 The School of Electronic and Information Engineering, Xi'an Jiaotong University, Xi'an, China

the server's available resources or task's resource requirement is misestimated. In this case, over-commitment results in resource contention and performance degradation.

A high-performance computing platform usually allocates resources based on CPU core utilization [2], while a cloud computing platform usually defines one or two types of fixed-quantity resources as a slot and then uses the slot as the allocation unit in resource management [3–8]. Both core-based and slot-based methods utilize coarse-grained resource allocations. However, the resource requirements of tasks are diverse [1,9–14]. Therefore, resource allocation based on fixed units leads to resource fragmentation and inefficiencies. Some research divides tasks into CPU-intensive and IO-intensive types based on the resource requirements of the task [15–17]. Furthermore, different types of tasks can run concurrently to reduce resource fragmentation. Some studies dynamically adjust the number of slots to avoid resource fragmentation. However, these methods result in inefficient resource sharing. Some inefficient sharing of resources, such as CPU, can lead to resource contention and seriously affect performance [18]. The over-commitment of certain resources, such as memory, directly leads to task failure or server crashes. In modern data centers, more than 53% of straggler tasks are caused by high resource utilization caused by inefficient resource sharing. Furthermore, 4–6% of total task straggling affects 37–49% of total jobs, resulting in considerable degradation of job completion times [19].

Certain resource scheduling algorithms [12–14,20,21] and some cloud computing platforms [22–24] use request-based mechanisms to allocate resources with fixed quantities to avoid resource fragmentation and over-commitment. However, resource fragmentation still occurs for the following reasons. On the one hand, resource requests may be specified manually; thus, there can be a difference between resource requests and actual usage [1]. On the other hand, a large gap between resource requests and actual usage exists when the resource request is equal to the maximum requirement [9]. Task resource usage fluctuates and does not continuously remain at the maximum. Scheduling that uses a request-based mechanism may have difficulty achieving high utilization rates [9]. For example, assume that a server has 4 CPU cores and 8 GB memory and that tasks A1, A2, B1 and B2 arrive at the server sequentially. The stage information format and actual requirements of tasks A and B are shown in Fig. 1. As the upper part of the figure shown, the task stage information includes CPU requirement, memory requirement and duration. And the detail information about stages of task A and task B is shown in the lower part of Fig. 1. Both tasks A and B have two stages.

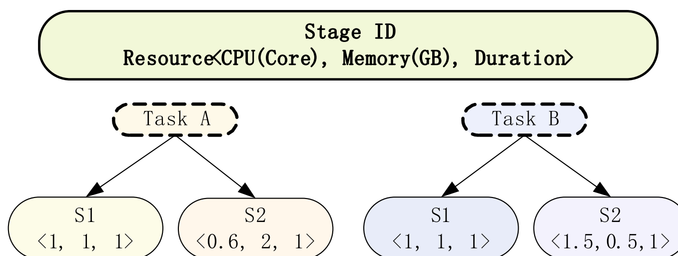

Fig. 1 Stages and requirements of tasks

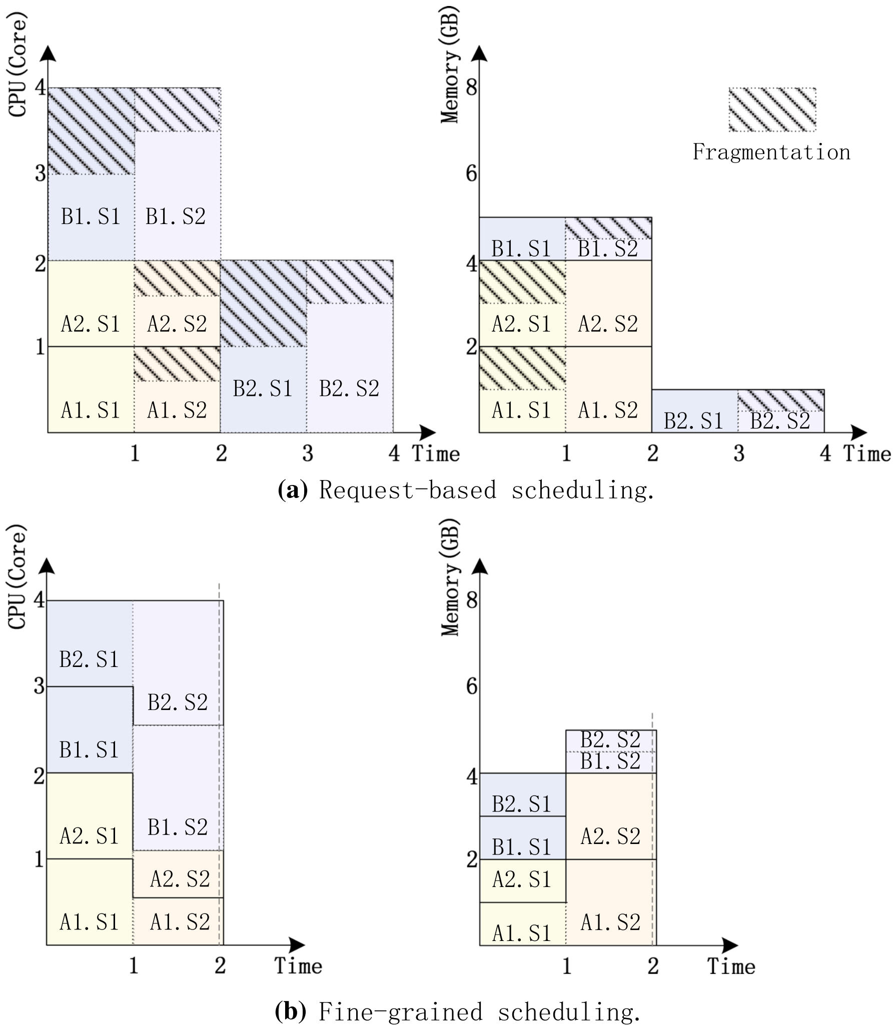

Fig. 2 Scheduling results of different methods

Task A uses 1 CPU core and 1 GB memory during the first time unit, and uses 0.6 CPU cores and 2 GB memory during the second time unit. Task B requires 1 CPU core and 1 GB memory during the first time unit, and 1.5 CPU cores and 0.5 GB memory during the second time unit. The allocation results of the request-based scheduling and fine-grained scheduling are shown in Fig. 2. And the diagonally hatched boxes in subgraph (a) of Fig. 2 represent resource fragmentation. Assume that task A uses its maximum resource usage to request resources and task B requests 2 CPU cores and 1GB memory. The CPU resources of the server are occupied after tasks A1, A2 and B1 are allocated requested resources. In this case, task B2 cannot receive resources until the other tasks complete. And actually, many free resources, which are covered with

the diagonally hatched boxes in the subgraph (a), are wasted in the form of resource fragmentation. The workload completes in 4 time units.

To address the above problems of resource allocation in multi-resource clusters, this paper proposes a fine-grained method (FGM) for scheduling and allocating resources. This method matches task resource requirements and available server resources by stages to refine the allocation granularity and avoid both resource fragmentation and over-commitment. During the allocation process, when necessary, this method compresses resource allocations according to both resource characteristics and resource availability to further improve resource utilization and performance. For the previous example, the allocation result of the FGM is shown in subgraph (b) of Fig. 2. The FGM estimates the actual resource requirement and divides the task into execution stages. Then, it matches the available resources for each execution stage according to the actual resource requirements and the duration of the execution stage. After tasks A1, A2 and B1 are allocated their requirement, 1 CPU core is free during the first time unit, and 1.3 CPU cores are free during the second time unit. The available server CPU is less than the resource requirement of task B2 when matching the second execution stage of task B2. In this case, the FGM compresses the CPU requirement (4.2 cores) into the CPU allocation (4 cores) of the second stage of all tasks after careful calculation, enabling all 4 tasks to run together. The slight compression improves resource utilization and the number of running tasks at the cost of extending completion time of all running tasks. As a result, the completion times of the second task stages increase, but the workload completion time decreases compared to the result in subgraph (a) of Fig. 2. The workload scheduled by the FGM takes 2.05 time units to complete. This result constitutes a large improvement in server resource utilization and workload completion time compared to request-based scheduling.

The main novel contribution of FGM is that FGM matches the resource supply and demand by stages to refine the resource granularity in allocation. This innovation can improve the matching degree between resource supply and demand, and effectively reduce the resource fragmentation and resource contention. The other novel contribution of FGM is that the controllable compression is innovatively introduced into resource matching and allocation. And this innovation further increases the number of running tasks and finally improves the resource utilization and performance. The main work of the FGM is summarized as follows.

- The FGM matches actual resource requirements and available resources by stages to achieve the fine allocation granularity of resource quantity and duration, which helps prevent resource fragmentation and over-commitment. Actual task resource requirements are estimated based on the runtime information from similar tasks. Task execution stages are calculated according to their actual resource requirements.
- Computing resources are divided into two categories—compressible and incompressible resources—according to their characteristics. The FGM slightly compresses allocations of compressible resources, if necessary, to improve the number of running tasks and resource utilization.
- To ensure resource utilization and performance, the FGM adjust resource compression and fine-grained scheduling using several mechanisms, including runtime

resource monitoring, allocation policy adjustments and scheduling constraint checks.

FGM was implemented as a pluggable scheduler based on Yarn [22]. The tests show that the FGM can resolve resource fragmentation and over-commitment problems and improve both resource utilization and performance with acceptable fairness and scheduling response times.

The remainder of this paper is organized as follows. Section 2 describes related works. Section 3 details the algorithms and mechanisms of the FGM, and Sect. 4 introduces the architecture and implementation of the FGM. A variety of test results are provided in Sect. 5. Section 6 provides conclusions and outlines ideas for future work.

2 Related work

Resource scheduling has been studied for decades. Here, the paper focuses on only the most relevant work for large-scale server clusters.

Hadoop [7] is an open-source cloud platform that uses a monolithic scheduler architecture. This platform divides the computing resources into slots that have fixed-quantity resources and allocates slots to MapReduce [26] tasks. However, the resource requirements of cloud computing workloads are diverse [1,9–14]. A slot used by Hadoop ignores resource requirement diversity and leads to wasted resources caused by resource fragmentation.

Yarn [22] is a cloud platform that is an evolved version of Hadoop. Yarn uses a two-level scheduler that separates resource and task management, which allows it to support diverse workloads. Fuxi [23] is a resource and job management system implemented by Alibaba. This system achieves scalability and fault tolerance by using mechanisms such as incremental resource management and failure recovery. Fuxi defines a ScheduleUnit used in resource requests. Borg [24] is the resource management system that Google has used to manage its large cluster. Borg allows users to request CPU resources in units of milli-cores and to request memory and disk space in bytes. However, Borg rounds resource allocation requests up to the next 0.5 cores for the CPU and 1 GB for memory. Borg classifies resources into compressible and incompressible resources in a manner similar to that described in the paper. However, Borg uses the classification only to determine the limit method of resource usage; it does not apply the classification to resource allocation.

Yarn, Fuxi and Borg use request-based mechanisms to determine resource allocation, and the allocated quantity is fixed during execution. The resource request may be specified manually or specified as a maximum resource usage [1]. However, resource usage fluctuates and does not remain at the maximum. Therefore, there is a gap between such resource requests and actual usage. This gap may cause request-based mechanisms to fail to achieve high resource utilization [9].

Mesos [27] is a resource management platform for cloud computing that was proposed by the AMP laboratory at the University of California, Berkeley. Mesos provides resource offers to application frameworks and allows frameworks to decide which

resources to accept and use. However, many application frameworks may not know the detailed resource requirements of tasks clearly. And the offer-based mechanism has the similar problems with request-based mechanism.

The method proposed in this paper matches the actual resource requirements and available resources by stages to resolve those problems. FGM can be implemented in any cluster or cloud platform to change the allocation granularity and avoid problems such as resource fragmentation and over-commitment.

Many studies have focused on scheduling algorithms. For example, Capacity [3] is a scheduler designed by Yahoo!. This scheduler divides computing resources into queues to allow users to share computing resources. Tetris [14] packs task resource requirements to resolve resource fragmentation and over-commitment problems caused by multi-resource requirements to improve cluster resource utilization. Compared with Tetris, the FGM divides execution stages according to their actual resource usage of task and considers compression when allocating resources to further improve cluster resource utilization.

Delay [5] schedules tasks to the servers that have input data to avoid network transmission and improve performance. Quincy [28] maps scheduling problems to a graph whose edge weights are data locality and fairness and then schedules tasks according to a global cost model. However, the scheduling latency of Quincy is extremely high with large clusters. Firmament [29] uses multiple min-cost max-flow algorithms and many other measures, such as incremental re-optimization, to resolve scheduling latency problems. Delay, Quincy and Firmament pay considerable attention to data locality and do not change the allocation granularity during scheduling. The FGM uses a data locality mechanism similar to that used by Delay [5] and considers only allocation granularity and resource utilization. LATE [6] and Mantri [30] resolve the task straggling problem by launching re-execute tasks. The FGM has no centralized speculation mechanism; each application is responsible for its own speculation strategy.

Apollo [12] is a coordinated scheduling framework proposed by Microsoft. This system analyzes resource availability based on task completion time and makes scheduling decisions based on resource availability. Omega [13] is a shared-state scheduler based on lock-free optimistic concurrency control [31]. Both Apollo and Omega can improve the quality and scalability of scheduling. However, resource fairness is a challenge for distributed scheduling in Apollo and Omega.

The Fair [4] scheduler provides resource fairness and data locality for multi-user clusters. The DRF [20] scheduler provides max–min fairness to multiple types. Choosy [32] is a scheduler that provides constraint max–min fairness. Carbyne [21] improves resource utilization at the cost of losing instantaneous fairness. The aim of the FGM is similar to that of Carbyne. However, the FGM does not propose any new fairness policies. Instead, it uses dominant resource fairness as the default policy and supports configuring multiple fairness policies. This approach improves resource utilization and performance by changing allocation granularity and compressing resource allocation.

The FGM focuses on matching resource requirements and available resources with fine granularity to improve resource utilization and performance. It compresses resource allocation according to resource characteristics and availability to further improve resource utilization and performance. Therefore, the FGM and the above methods are not in conflict but complementary. They can work together to improve

scheduling decisions. For example, the DRA method [33] can use the FGM to improve the allocation granularity and matching degrees when it allocates resources in the federated cloud environment. As another example, the list-scheduling algorithm proposed in [34] ignores the fact that task requirements change constantly when scheduling tasks to heterogeneous processors. The resource fragmentation can be effectively reduced if the algorithm uses the FGM to address the changing resource requirements and improve the allocation granularity when scheduling.

There are two approaches to quantify resource requirements and other task information. One uses historical workload information to provide estimates [14,35–37]. The other uses similar tasks to estimate information [12,38–40]. The FGM estimates information related to resources according to similar tasks. Compared to existing research, the estimation in FGM has the following differences. First, the estimation in the paper uses an iterative calculation to mitigate the effects of bad data on estimation results. This process smoothes the curve of the estimation results. Second, because resource compression may change the accuracy of runtime information, FGM dynamically adjust the weight of the runtime information in iteration based on the compression rates to ensure the accuracy of resource estimation.

3 Fine-grained method of resource scheduling

The FGM refines allocation granularity in two aspects. First, the FGM refines the granularity of allocation duration by performing resource matching by stages rather than according to a fixed value or the whole duration of a task. Second, the FGM refines the granularity of allocation quantity by basing allocations on the actual requirement of the execution stage rather than the requested quantity, the maximum requirement, a slot, or another unit. In addition, FGM makes further improvement in running tasks number and resource utilization by introducing controllable compression into resource matching and allocation.

3.1 Compression of computing resources

Computing resources have different characteristics. For some computing resources, task duration will be extended without affecting the task completion success if the allocation is less than the requirement. For example, assume a task needs 1 core but its allocation is half of a core. In this case, the task can still be completed successfully in twice the original completion time. For other computing resources, tasks cannot be successfully completed when the resource allocation is less than the requirement. For example, suppose a task requires 1 GB to save data. The task will fail if the allocation is less than 1 GB. The FGM divides computing resources into two categories-compressible and incompressible-based on the resource characteristics. Furthermore, the FGM defines the compression rate to measure the degree of resource compression.

Definition 1 (*Compressible and incompressible resources*) Suppose that the allocation of a resource is less than the task requirement. If the task can be completed successfully

by extending its duration, the resource is compressible; otherwise, the resource is incompressible.

Definition 2 (*Compression rate*) Suppose that the resource requirement is R_{r} and the allocation is R_{u}. The compression rate r_{c} can be calculated using (1) if the resource is compressible. The compression rate of an incompressible resource is always 0.

$$r_{\mathrm{c}} = \begin{cases} (R_{\mathrm{r}} - R_{\mathrm{u}})/R_{\mathrm{r}}, & R_{\mathrm{r}} > R_{\mathrm{u}} \\ 0, & R_{\mathrm{r}} \leq R_{\mathrm{u}} \end{cases} \tag{1}$$

The compression rate of compressible resource is calculated if resource requirement is larger than resource allocation. And the value of compression rate is equal to the ratio of total resource shortfall to total resource requirement.

Resource compression increases the resource contention between tasks and leads to an increase in task execution time. And a serious resource contention may result in lots of influences including great performance degradation and services' failure rate increase. Therefore, the compression rate should be limited. Suppose that the amount of a resource in a server is 1 and $n+1$ tasks are waiting for a resource. The resource requirement percentage of the ith task is μ_i, and its work quantity during time T is w_i. The task requirements satisfy $\sum_{i=1}^{n} \mu_i < 1$ and $\sum_{i=1}^{n+1} \mu_i > 1$. For task m, the total resource usage is fixed when the task finishes the quantity of work w_m. Then, the relationship shown in (2) in which T_{t}' is the task's execution time with compression and r_{c} is the compression rate when $n+1$ tasks are executed together can be obtained.

$$T \times \mu_m = T_{\mathrm{t}}' \times \mu_m \times (1 - r_{\mathrm{c}}) \tag{2}$$

The formula for calculating T_{t}' is shown in (3) which can be deduced from (2).

$$T_{\mathrm{t}}' = \frac{T}{1 - r_{\mathrm{c}}} \tag{3}$$

In reality, the real execution time with compression T' is larger than T_{t}' for many reasons, including resource contention. T' can be calculated using Formula (4), in which Δp represents the performance change caused by compression:

$$T' = (1 + \Delta p) \times T_{\mathrm{t}}' = (1 + \Delta p) \times \frac{T}{1 - r_{\mathrm{c}}}. \tag{4}$$

The goal of resource compression is to improve resource utilization and overall workload performance at the cost of slightly extending the completion time of tasks using the resource. Therefore, more work can be finished per unit time after compression. The relationship shown in (5) can be obtained.

$$\frac{\sum_{i=1}^{n} w_i}{T} < \frac{\sum_{i=1}^{n+1} w_i}{T'}. \tag{5}$$

Then, the constraint of the maximum compression rate during time T shown in Eq. (6) can be obtained by substituting Eq. (4) into Relationship (5):

$$r_c < \frac{w_{n+1} - \sum_{i=1}^{n} w_i \times \Delta p}{\sum_{i=1}^{n+1} w_i}. \tag{6}$$

The maximum compression rate is nonnegative. Therefore, the FGM calculates the maximum compression rate during a time period using Formula (7) to ensure that the resource utilization and overall performance of workloads will be improved after compression.

$$r_c = \max\left(0, \frac{w_{n+1} - \sum_{i=1}^{n} w_i \times \Delta p}{\sum_{i=1}^{n+1} w_i}\right) \tag{7}$$

The maximum compression rate calculated by Formula (7) is used to provide dynamic limitation at runtime. The resource compression rate is also limited by a fixed limitation configured by administrator. And the fixed limitation of resource compression rate should never be exceeded at runtime. The appropriate limitation is related to many factors such as workload, hardware configuration of server and computing environment. Therefore, the value of limitation should be determined by many tests. The default limitation of resource compression rate is 10% in FGM.

3.2 Resource requirement estimation

The basis of resource matching in an FGM is to accurately estimate the requirements and duration of all types of resources. Workloads designed for large-scale computing clusters often have hundreds of thousands of tasks that have similar resource usages. This section first defines similar tasks based on those that have similar execution logic. Then, the section uses these similar tasks to estimate the task resource requirements for future tasks.

Definition 3 (*Similar tasks*) Let L represent the execution logic of a task and D represent the size of the input data that the task processes. If there is a relationship such that $L_i = L_j \wedge D_i = D_j$ between task i and task j, then the two tasks can be designated as similar tasks.

Definition 3 only constrains the execution logic and input data size of similar tasks. This definition is useful for the estimation results reuse of different jobs that have the same execution logic. The FGM uses Formula (8) to iteratively calculate the actual resource requirements and duration of a certain process. For resources such as the IO bandwidth of disks and networks, the calculations are performed according to the relationship between the locations of data and tasks. The location relationships include data and tasks hosted on the same server, on the same rack, and other situations.

$$\alpha_n = \begin{cases} \text{average}(\beta_n, \beta_{n+1}, \beta_{n+2}), & n = 1 \\ e^{\min(\text{Th}_r \times r_c - 5/60, 0)} \times \alpha_{n-1} + \left(1 - e^{\min(\text{Th}_r \times r_c - 5/60, 0)}\right) \times \beta_{n+2}, & n \geq 1 \end{cases} \tag{8}$$

In Formula (8), α_n is the nth estimated result, β_n is the nth reported information, r_c is the compression rate of the reported information, and Th_r is a parameter that limits the maximum compression rate. This formula refers to the calculation of average resource utilization in the Linux system. The iterative calculation can mitigate the effects of inaccurate data on estimation results. The accuracy of similar task runtime information declines when resource allocation is compressed. Therefore, in order to ensure the effectiveness of the estimation results, the FGM dynamically adjusts the weights for reported information in its iterative calculation based on the compression rate of the reported information. The weight of reported information is 0 if the compression rate is greater than $5/(60 \times \text{Th}_r)$. In this case, the reported information does not affect the estimation results.

The FGM divides tasks into several execution stages according to the actual requirements of the tasks and updates the division of the execution stages at a regular interval. The flowchart of execution stage division is shown in Fig. 3, in which P, P_s and P_e represent the progress being calculated, the starting progress of the execution stage and the ending progress of the execution stage, respectively. $C_{\max}$, $C_{\min}$, $M_{\max}$ and $M_{\min}$ are the maximum and minimum CPU and memory resources used during the execution stage, respectively. Th_c, Th_m and Th_p are the division thresholds of CPU,

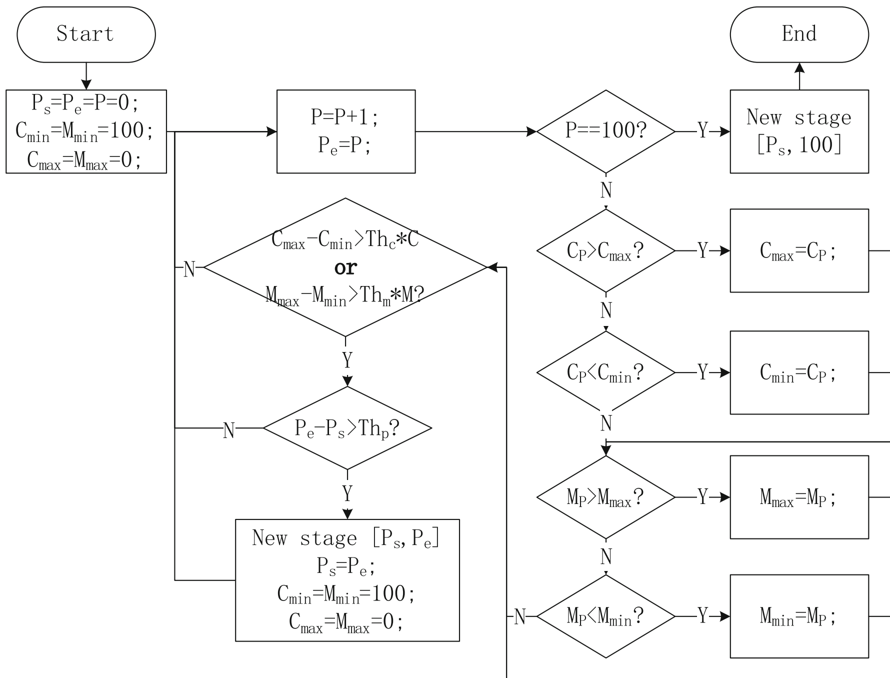

Fig. 3 Flowchart of execution stage division

memory and progress, respectively. $[P_s, P_e]$ is partitioned into an execution stage when the duration of the execution stage reaches a threshold and the fluctuations in the actual CPU or memory requirements exceed a certain level. For resources such as CPU and memory, the actual requirement of the execution stage is the maximum requirement among all progresses within the execution stage. For a resource such as disk or network bandwidth, the actual requirements of the execution stage are the average requirements of all progresses within the execution stage.

3.3 Fine-gained scheduling algorithms

In this paper, to simplify the algorithm expression, certain variables related to resources are expressed as tuples. The tuple shown in Eq. (9) represents a set of ordered values, and R^i is the ith value of the tuple.

$$\overrightarrow{R} = \left\langle R^1, \ldots R^n \right\rangle \tag{9}$$

Suppose that $\overrightarrow{R_a} = \langle R_a^1, \ldots R_a^n \rangle$ and $\overrightarrow{R_b} = \langle R_b^1, \ldots R_b^n \rangle$. Then, the two tuples can be added or multiplied using Formulas (10) and (11), respectively, while a tuple can be multiplied by a constant using Formula (12).

$$\overrightarrow{R_a} + \overrightarrow{R_b} = \left\langle R_a^1 + R_b^1, \ldots R_a^n + R_b^n \right\rangle \tag{10}$$

$$\overrightarrow{R_a} \times \overrightarrow{R_b} = \left\langle R_a^1 \times R_b^1, \ldots R_a^n \times R_b^n \right\rangle \tag{11}$$

$$c \times \overrightarrow{R_a} = \left\langle c \times R_a^1, \ldots c \times R_a^n \right\rangle \tag{12}$$

The relationship $\overrightarrow{R_a} < \overrightarrow{R_b}$ is evaluated according to Formula (13). In other words, $\overrightarrow{R_a}$ is less than $\overrightarrow{R_b}$ if any element in $\overrightarrow{R_a}$ is less than the corresponding element in $\overrightarrow{R_b}$:

$$(\exists i) \left(0 < i < n \wedge i \in Z \wedge R_a^i < R_b^i \rightarrow \overrightarrow{R_a} < \overrightarrow{R_b}\right). \tag{13}$$

The consideration the comparison rule shown as Formula (13) is the matching failure in scheduling. Suppose there are the available resource tuple of server and the resource requirement tuple of task. The resource matching fails if any available quantity of resource does not satisfy task requirement of this resource.

Fine-grained scheduling matches and allocates resources when free resources appear on the servers. The details are shown in Algorithm 1. To start, the algorithm checks the idle resources and scheduling policy of the server to decide whether to continue scheduling (lines 1–3). The elements in the tuple of free resources on the server can be calculated using Formula (14), in which r is the value of the element, r_i is the resource quantity of the ith sample value, t_i is the duration of the ith sample and T is the total time. On the basis of resource sample information, this formula uses time as the weight to calculate the value of free resource quantity.

$$r = \sum_{i=1}^{n} (r_i \times t_i)/T. \tag{14}$$

After the resource check, the algorithm sorts set T_s, the tasks waiting for scheduling, using the fairness policy P_f to ensure resource fairness (line 4). Then, the algorithm traverses set T_s to check data locality and match available resources. The matching strategy is determined by both the estimation result of the task and the server's scheduling policy. The algorithm matches the available resources based on resource requests when the estimation results are insufficient or when the task requirements cannot be estimated (lines 8–9). The algorithm matches the resource requirement and available resources using the fine-grained matching algorithm if the scheduling policy of a server is fine grained (lines 10–11); otherwise, it matches available resources using a coarse-grained strategy (lines 12–18). For the coarse-grained matching, the resource requirement tuple $\overrightarrow{M_x}$ consists of $\overrightarrow{M_{max}}$ and $\overrightarrow{M_{avg}}$. $\overrightarrow{M_{max}}$ includes the resource requirements that need to be allocated using the maximum requirements, such as CPU and memory space, while $\overrightarrow{M_{avg}}$ includes the resource requirements that can be allocated using average requirements, such as disk and network bandwidth. The available resource tuple $\overrightarrow{A_x}$ consists of $\overrightarrow{A_{min}}$ and $\overrightarrow{A_{avg}}$. $\overrightarrow{A_{min}}$ includes the minimum available quantity of a resource that must be allocated using the maximum requirement over the interval $[0, T]$ (assuming the current time is 0 and the task duration is T), while $\overrightarrow{A_{avg}}$ includes the average available quantity of a resource that can be allocated by the average requirements over the interval $[0, T]$. The matching fails if any requirement is not satisfied ($\overrightarrow{A_x} < \overrightarrow{M_x}$). If the matching succeeds, the algorithm calls checkQoS() to check whether the allocation affects the quality of service of all running server tasks (line 19). If all checks pass, the algorithm allocates the resources to a task and analyses the remaining resources to determine whether to enter the next round of scheduling (lines 20–24).

In the FGM, the default logic of checkQoS() checks the task completion time when the resource allocation of the server is compressed. To ensure the service quality of tasks running on the server, this check analyzes the completion time for all running tasks on the server by assuming that resources are already allocated to the task. The task's theoretical completion time D_t can be calculated using Formula (15), in which D_i is the duration of the ith stage without compression and r_{ci} is the maximum compression rate of resources in the ith stage:

$$D_t = \sum_{i=1}^{n} (D_i/(1 - r_{ci}). \tag{15}$$

The fine-grained matching algorithm shown in Algorithm 2 traverses all the task execution stages and matches available resources and requirements while considering the compression characteristics of the various resources. The matching is successful when all the resource requirements are satisfied over all the task execution stages (lines 4–20). During the matching process, the available tuple of a compressible resource $\overrightarrow{n_c}$ and the available tuple of an incompressible resource $\overrightarrow{n_n}$ are compared with require-

Algorithm 1 Fine-grained scheduling

Input: T_s: the set of tasks waiting for scheduling.
P_a: the schedule policy of server.
$\overrightarrow{I}$: the free resource tuple of the server.
$\overrightarrow{N}$: the total resource tuple of the server.
Output: T_A: the set of scheduled tasks.

1: **if** $\overrightarrow{I} < \overrightarrow{\text{Th}} \times \overrightarrow{N}$ **or** isUnschedulable(P_a) **then**
2: cannot allocate resource of the server, just return;
3: **end if**
4: sort(T_s, P_f);//sort the task set by the fairness policy
5: **for** t in T_s **do**
6: check the data locality of task t;
7: match ← **true**;
8: **if** unpredictable(t) **then**
9: match ← allocRequestQuantity();//task t is a service or there is not enough information for task t
10: **else if** P_a==FineGrained **then**
11: match ← fineGrainedMatching(); //the algorithm is shown in Algorithm 2
12: **else**
13: $\overrightarrow{M_x} \leftarrow \left\langle \overrightarrow{M_{\max}}, \overrightarrow{M_{\text{avg}}} \right\rangle$;
14: $\overrightarrow{A_x} \leftarrow \left\langle \overrightarrow{A_{\min}}, \overrightarrow{A_{\text{avg}}} \right\rangle$;
15: **if** $\overrightarrow{A_x} < \overrightarrow{M_x}$ **then**
16: match ← **false**;
17: **end if**
18: **end if**
19: **if** match **and** checkQoS() **then**
20: allocate(t); //allocate resource to task t
21: $T_A \leftarrow T_A \cup t$;
22: **if** the resource is used up **then**
23: **break**;//only check the resource quantity before next heartbeat
24: **end if**
25: **end if**
26: **end for**

ments, respectively. For incompressible resources, the algorithm directly compares the requirement tuple with the available resource tuple. For compressible resources, the algorithm compares the requirement tuple with the sum of the available resource tuple and the product of the total quantity of compressible resources $\overrightarrow{N}$ and the limitation of the maximum compression rate $\overrightarrow{r_c}$ in this stage. The values in $\overrightarrow{r_c}$ can be calculated using Formula (7) based on the task information, the running tasks on the server and the scheduling policy of the server.

3.4 Runtime resource monitoring

A gap between estimated resource requirements and actual resource usage is inevitable because resource usage is affected by many factors. The FGM defines "estimation deviation" to measure the degree of deviation between estimated resource requirements and actual resource usage and saves recent estimation deviations of each resource for future use in adjusting the scheduling policy.

Algorithm 2 Fine-grained matching

Input: n_s: the available resource stages of server.
t_s: the execution stages of task t.
$\overrightarrow{N}$: the total resource tuple of the server.
Output: the matching succeed or fail.

```
1: n ← the first stage in n_s;
2: n_c ← the tuple of the compressible resource of stage n;
3: n_n ← the tuple of the non-compressible resource of stage n;
4: for s in t_s do
5:     s_c ← the tuple of the compressible resource of stage s;
6:     s_n ← the tuple of the non-compressible resource of stage s;
7:     while n.startTime < s.endTime do
8:         if n_c + r_c × N < s_c or n_n < s_n then
9:             return false; //not match
10:        end if
11:        if n.endTime ≤ s.endTime then
12:            n ← the next stage in n_s;
13:            n_c ← the tuple of the compressible resource of stage n;
14:            n_n ← the tuple of the non-compressible resource of stage n;
15:        else
16:            break;
17:        end if
18:    end while
19: end for
20: return true;
```

Definition 4 (*Estimation deviation*) The estimation deviation F of the resource on one server can be calculated using Formula (16), in which μ_i is the actual resource usage of the ith stage, α_i is the estimated resource requirement of the ith stage, t_i is the duration of the ith stage, T is the total duration of all stages, μ is the average actual resource usage, α is the average estimated resource requirement, I is the average free quantity of this resource, Th is a threshold of free resources and N is the total quantity of this resource. The values of μ, α and I can be calculated using Formula (14).

$$F = \begin{cases} \sum_{i=1}^{n} ((\mu_i - \alpha_i)^2 \times t_i)/T, & \mu < \alpha \text{ and } I < \text{Th} \times N \\ 0, & \mu \leq \alpha \text{ or } I \geq \text{Th} \times N \end{cases} \tag{16}$$

Based on the definition above, the estimation deviation of a resource can be nonzero only when the actual resource usage is larger than the estimated resource requirement and the average free quantity of this resource is below a certain threshold. Improving the estimation result accuracy requires an evaluative process; therefore, the FGM allows the estimation mechanism to correct the estimation results when the gap between the estimated resource requirements and actual resource usage will not lead to negative consequences.

3.5 Scheduling policy adjustment

Problems such as inaccurate estimation and inappropriate compression may lead to incorrect allocation decisions and result in excessive resource contention. Therefore, the FGM dynamically adjusts the scheduling policy of the server to mitigate the effects of incorrect scheduling decisions according to recent estimation deviations of server resources and the completion times of server tasks. The threshold of estimation deviation when making scheduling policy adjustments is calculated by Formula (17), in which N is the total quantity of the resource, $\mathrm{Th_t}$ is the time threshold and $\mathrm{Th_f}$ is the fluctuation degree threshold:

$$\mathrm{Th}_F = \mathrm{Th_t} \times (\mathrm{Th_f} \times N)^2. \tag{17}$$

The FGM changes $\mathrm{Th_t}$ and $\mathrm{Th_f}$ to generate several thresholds to measure the deviation extents of recent estimation deviations of servers. The scheduling policy of the server can be adjusted to be fine-grained, fine-grained without compression, coarse-grained, to stop resource allocation or to kill tasks to recycle resources according to the extent of the resource contention problem on the server.

4 Architecture and implementation of the fine-grained scheduler

4.1 Architecture of the fine-grained scheduler

Based on the FGM, the paper designed the fine-grained scheduler whose architecture is shown in Fig. 4. The schedule core at the center of the scheduler coordinates with the other modules to complete scheduling according to the algorithms. The estimator module is responsible for processing the runtime information reported by the servers, estimating the actual resource requirements of tasks and the resource availability of servers, and dividing the execution stages of tasks. The estimation results are asynchronously updated, and the module does not guarantee that the estimation result will include the latest information reported from the servers. This asynchronous mechanism greatly reduces the impacts of information processing, requirement estimation and execution stage dividing on the time required to access the estimation results. The strategy manager is responsible for managing strategies such as resource matching and fairness, and providing strategy support to the schedule core according to the scheduling policy of the server and other configuration information. In addition, the strategy manager allows strategies to be added or changed at runtime. The constraint checker ensures that the service qualities of all running tasks can be satisfied after the new allocation decision goes into effect. Furthermore, constraint checking and resource matching work simultaneously to improve scheduling efficiency. The constraint checking stops working directly when resource matching or other checks fail. The strategy adjuster saves the recent server estimation deviations and dynamically adjusts the scheduling policy of the server according to the deviations and the recent task completion times of the server. The compressibility manager analyzes the limi-

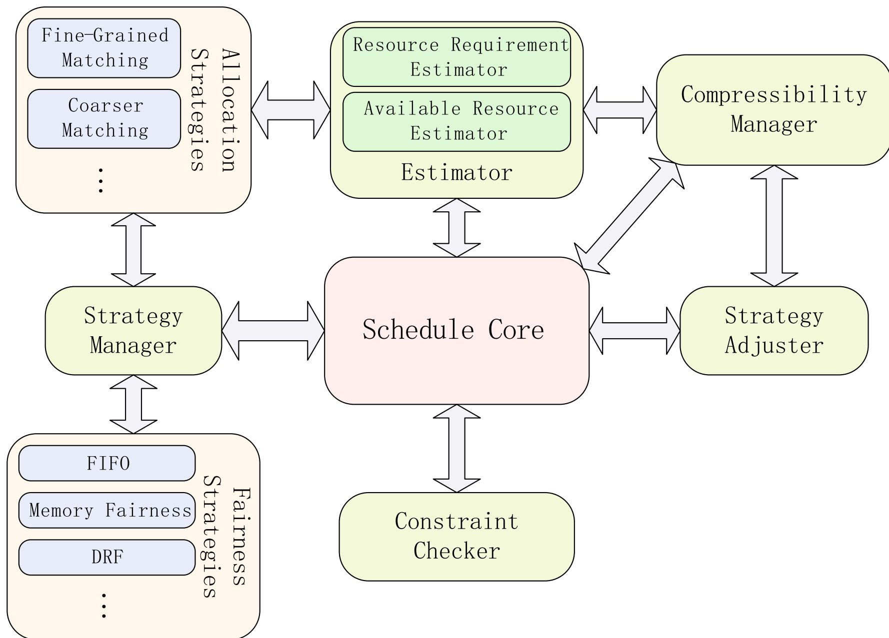

Fig. 4 Architecture of the fine-grained scheduler

tations of the maximum compression rate of the server according to task information and scheduling policies obtained from the estimator and the strategy adjuster.

4.2 Implementation of the fine-grained scheduler

The fine-grained scheduler was implemented in Yarn platform. The scheduler modules are shown in Fig. 5. The implementation added an analysis module to the node manager. This module processes files in the Proc directory to collect information from the running tasks and servers. The node manager uses the collected information to calculate the estimation deviations and free resources. All the information is periodically reported to the resource manager. The application master sends the MD5 value of the application code and the input data size to the resource manager when the application master registers itself. The application master uses its identity and task-type information to request resources from the resource manager. The application master and the task-type information help the estimator identify the estimation results. The scheduler and estimator run as services of the resource manager. The estimator is responsible for the work related to estimation. The scheduler includes independent modules such as the strategy manager, strategy adjuster and constraint checker. The scheduler is responsible for scheduling-related work. The data-locality mechanism of the scheduler is very similar to that used in Delay [5]. The default fairness policy in the implementation is dominant resource fairness [20].

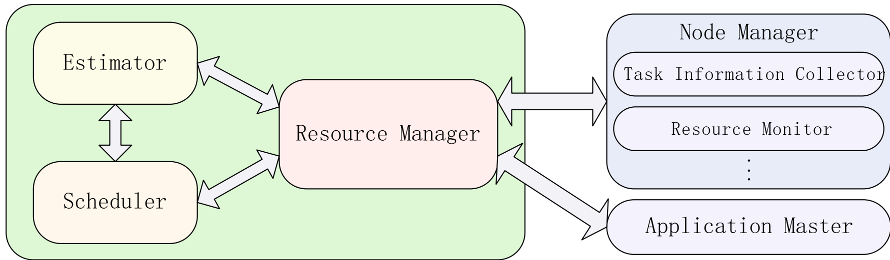

Fig. 5 Diagram showing the implementation of the fine-grained scheduler in the Yarn platform

The fine-grained method can also be used in other cluster platforms to improve resource utilization and performance.

5 Experiments and evaluation

5.1 Introduction to experimental environment and workload

The experiments were performed on a cluster at the China National High-Performance Computing Center in Xi'an. The cluster contained 24 servers with the configuration shown in Table 1. The paper deployed the Yarn platform on the cluster and configured numerous frameworks on the platform, including MapReduce and Spark. The workload consisted of many different jobs, and the data sizes processed by the jobs were similar to the data sizes of the jobs in Facebook [4]. The input data came from Wikipedia and from random generation.

The paper compared the FGM with several other algorithms, including FIFO, Capacity [3], Fair [4] and DRF [20], using both offline and online workloads. The offline workload submitted all jobs at its start, and its total input size was 400 GB. For the online workload, the times at which the jobs were submitted followed a Poisson distribution. The experiments were performed in dedicated, typical cloud and resource contention environments. In the dedicated environment, the cluster ran only the Yarn workload. In the cloud environment, the cluster ran both the Yarn workload and scientific computing applications, a situation that is similar to the real-world state of a typical cluster. In the resource contention environment, the paper introduced more

Table 1 Cluster server configurations

CPU	Memory	Storage	Network
2 × Intel Xeon	8 × 4 GB	2 × 300 GB	2 × 1000 Mbps and infiniBand QDR HCA 40 Gbps
E5-2670@2.6 GHz	REG ECC		
(8 Cores, 16 Threads)	(DDR3 1600 MHz)	(10 krpm SAS)	

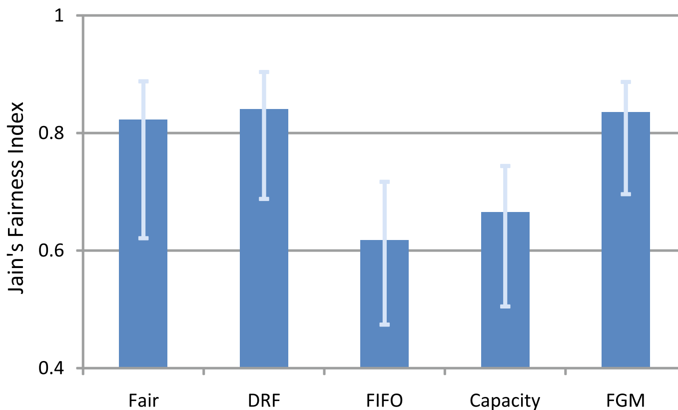

Fig. 6 Resource fairness of the offline workload in the dedicated environment

scientific computing applications than were running in the cloud environment and increased the input data size to 800 GB.

In the evaluation, the paper used Jain's fairness index [25] as the metrics of resource allocation fairness. The workload completion time is the time required to finish all tasks and starts at the point when the workload begins to submit. The scheduling response time covers the period from receipt of the allocation request to successful matching. This response time includes both the wait time and the resource matching time.

5.2 Experiments in the dedicated environment with the offline workload

The average fairness of the offline workload in the dedicated environment is shown in Fig. 6, in which error bars indicate the maximum and minimum fairness. In the figure, the average fairness of the DRF is best, and the average fairness of the FIFO is the worst. The average fairness of the FGM is the second best but only slightly below the result of the DRF. The FGM sorts the task set waiting for scheduling using the fairness policy to ensure allocation fairness. Compared to other algorithms, the FGM matches more resource types, and its matching granularity is finer. The results in Fig. 6 demonstrate that the FGM can achieve good resource fairness of the offline workload in the dedicated environment.

The offline workload submitted all jobs simultaneously, and the average scheduling response time of the offline workload is larger than that of the online workload. Therefore, the paper used the offline workload to test the scheduling response times of the algorithms. The average scheduling response times are shown in Fig. 7. As illustrated, the average scheduling response time of FIFO is much higher than that of the other algorithms. The FIFO is short for "first in first out". The FIFO algorithm schedules jobs according to job submission times. The later jobs wait a long time if resources are occupied by large job. Therefore, the high scheduling response time of FIFO algorithm is related to its allocation policy. The average scheduling response time of the Capacity algorithm is the second highest among all the algorithms. The reason is that the default policy inside a queue in Capacity algorithm is similar to the policy of FIFO

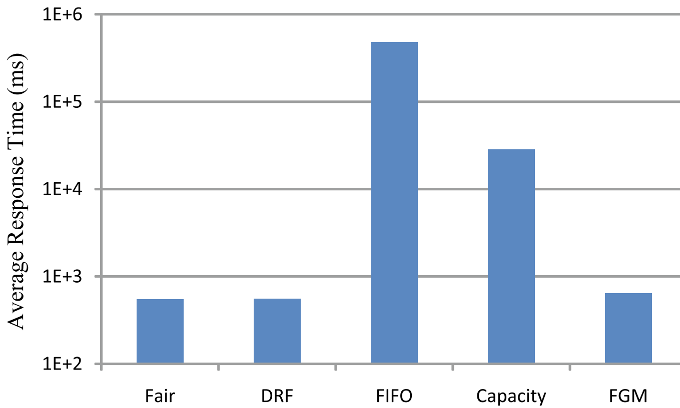

Fig. 7 Average response time of scheduling in the dedicated environment using the offline workload

policy. The average scheduling response time of the FGM is much lower than that of both the FIFO and Capacity algorithms. Compared to the Fair and DRF algorithms, the increment of the average scheduling response time of the FGM is less than 100 ms. The mechanisms of the FGM, such as information processing, requirement estimation and resource matching, work asynchronously. Therefore, the increment of scheduling response time is mainly caused by the two reasons. On the one hand, the fine-grained matching algorithm takes disk and network bandwidth into consideration. And the number of matching resource type is larger than other algorithms. On the other hand, the fine-grained matching algorithm improves the allocation granularity and increases scheduling cost. The scheduling response time increment of the FGM is acceptable if it can improve resource utilization and performance compared to the Fair and DRF algorithms.

The average task completion times of the offline workload in the dedicated environment are shown in Fig. 8, in which the blue and green columns indicate the average completion times for small and large tasks, respectively, and the red columns indicate the average completion times for all tasks. As shown, the average completion time of small tasks under the FGM is larger than the average completion time of small tasks under the other algorithms; the increments are between 0.62 and 4.17 s. This result arises from two causes. First, the FGM may compress task resource allocations during scheduling. Second, compared to large tasks, small tasks are more sensitive to resource compression due to their shorter execution time. In addition, both the increment of matching resource type number and improvement in allocation granularity increase the computing cost of FGM. Therefore, the other scheduling method or FGM with fine-grained matching and resource compression disabled may be more suitable considering computing cost and resource compression if the workload is dominated by small jobs. The average completion time of the large tasks under the FGM is less than the average completion time of large tasks under the other algorithms; the reductions are between 13.87 and 67.05 s. This result occurs for the following reasons. First, large tasks are insensitive to resource compression because they have longer execution times and they are not always compressed. Second, the FGM matching algorithm takes

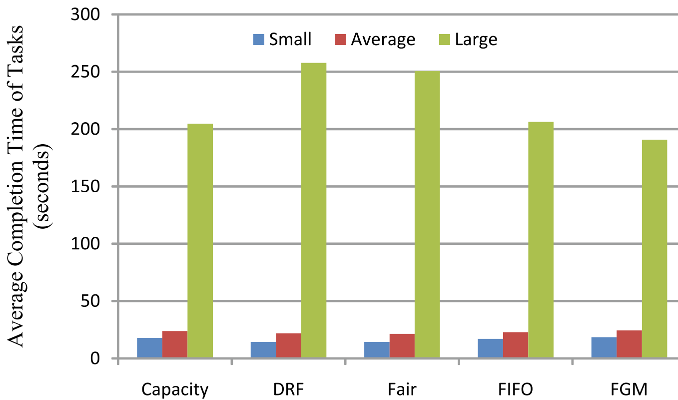

Fig. 8 Average completion times of tasks in the dedicated environment using the offline workload

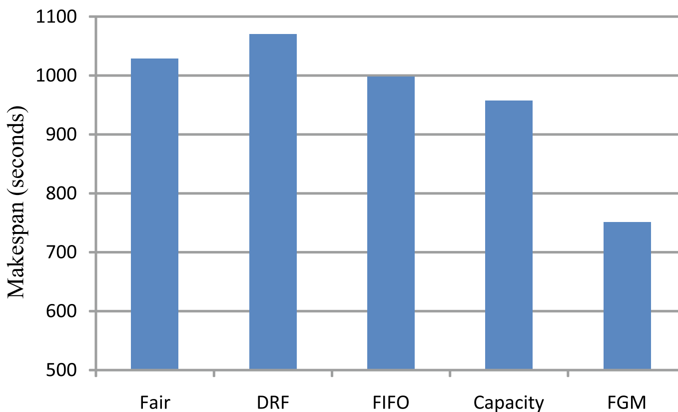

Fig. 9 Completion times of offline workloads in the dedicated environment

more resource types into consideration and refines resource allocation granularity to improve the allocation matching degree. In addition, the FGM limits the compression rate, monitors resource usage, checks the scheduling constraints and adjusts scheduling policies to avoid serious resource contention. The average completion time of all tasks under the FGM is slightly increased over the average completion time for all tasks under the other algorithms; the increments are between 0.5 and 2.88 s. However, this increase in the average completion time for all tasks does not necessarily indicate performance degradation because, to a certain extent, the improvement in resource matching granularity and compression in resource allocation may lead to the increase in the number of tasks can be executed on one server.

The completion times of offline workloads in the dedicated environment are shown in Fig. 9. These experimental results show that the FGM achieved a performance improvement ranging from 21.54 to 29.81% compared with the other 4 algorithms. Consequently, the FGM can effectively improve server resource utilization and improve the offline workload performance in the dedicated environment.

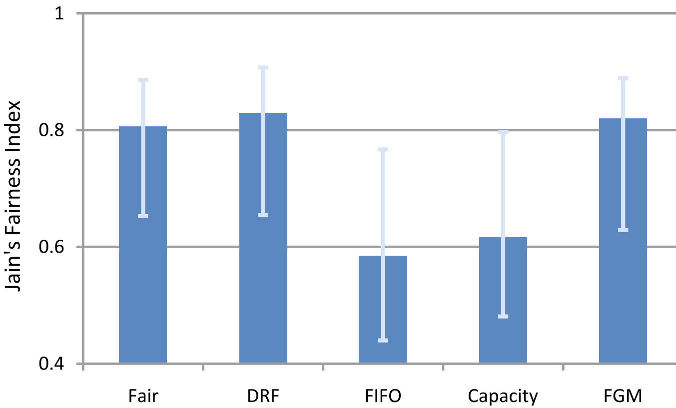

Fig. 10 Resource fairness of the online workload in the dedicated environment

5.3 Experiments in the dedicated environment with the online workload

The average fairness of the online workload in the dedicated environment is shown in Fig. 10. Compared to the results shown in Fig. 6, the fairness values achieved by the algorithms declined slightly because the job submission times of the online workload follow a Poisson distribution. In this test, the average fairness of the FGM algorithm is 1.19% lower than that of the DRF algorithm but higher than the average fairness of the other algorithms. Consequently, the FGM achieves good resource fairness for the online workload in the dedicated environment.

Figures 11 and 12 show the average job completion times and the distributions of job completion times for the compared approaches using the online workload. Compared with FIFO and Capacity, the FGM achieves an improvement in average job completion time of approximately 30%, while its improvement is approximately 3% compared to Fair and DRF. As can be seen from the job completion time distributions, compared with Fair and DRF the FGM has no obvious advantage when the job completion time is less than 150 s. However, for the jobs with completion times longer than 150 s, the advantage of the FGM become increasingly obvious. Furthermore, the job completion rate of the FGM surpasses those of the other algorithms after the job completion times exceed 250 s. The maximum job completion time achieved by the FGM is approximately 700 s, considerably better than the compared approaches. The workload completion time is correlated with longer job completion times because the workload completion has a long tail. Therefore, reducing the completion times for large jobs is more beneficial to performance improvements.

The completion times of online workloads in the dedicated environment are shown in Fig. 13. From these graphs, it can conclude that the improvement achieved by the FGM increases as the workload data size increased. The improvement eventually reaches approximately 30%. This result may occur for the following reasons. First, the number of large jobs is related to the workload input data size. When the workload input size increases, the number of large jobs also increases. Second, the FGM achieves greater improvements on large jobs than on small jobs. The results of this

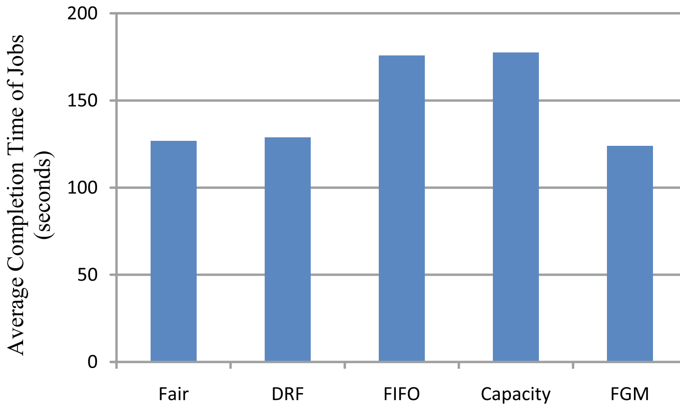

Fig. 11 Average completion time of jobs in the dedicated environment using the online workload

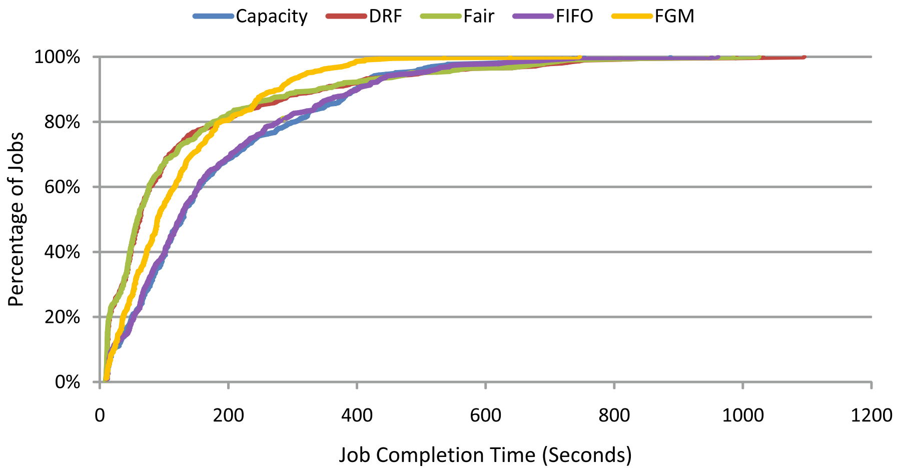

Fig. 12 CDF of job completion times in the dedicated environment using the online workload

test demonstrate that the FGM can improve resource utilization and performance in the dedicated environment.

5.4 Experiments in the cloud environment with the online workload

The average task completion times of the online workload in the cloud environment are shown in Fig. 14 (the column meanings are the same as in Fig. 8). As the results show, the average small task completion time of the FGM is longer than that required by the Fair and DRF algorithms, and the increment is approximately 1.6 s. The average small task completion time of the FGM is lower than that of the FIFO and Capacity algorithms, and the maximum reduction is 5.91 s. Second, the average large task completion time of the FGM is significantly less than that of the compared algorithms, and the improvements are between 36.36 and 137.12 s. Finally, the average com-

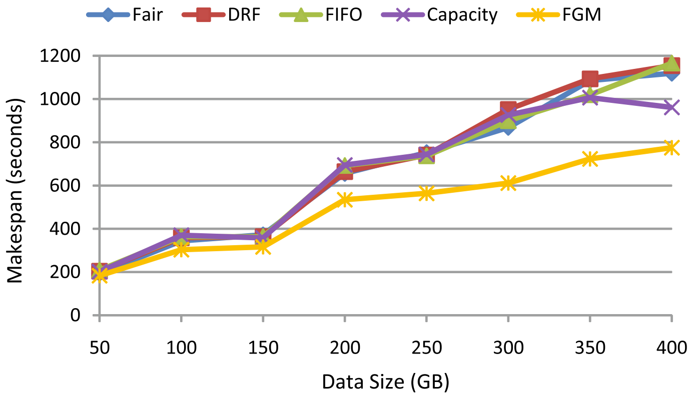

Fig. 13 Completion times of online workloads in the dedicated environment

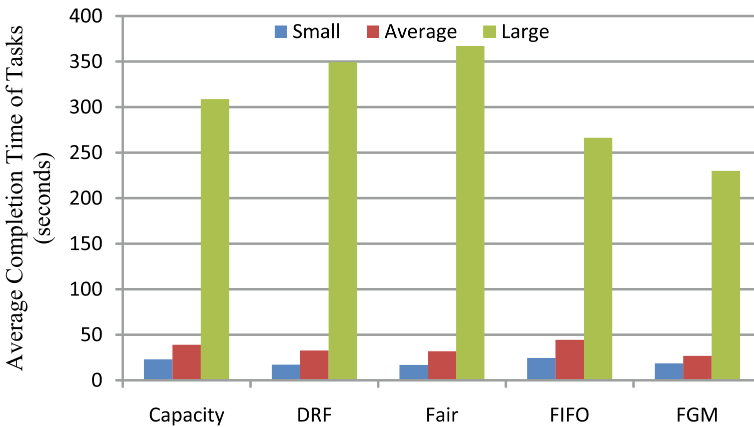

Fig. 14 Average completion time of tasks in the cloud environment using the online workload

pletion time of all tasks of the FGM is less than those of other algorithms, and the improvements are between 5.02 and 17.67 s. Note that the test results for the average task completion time in the cloud environment are significantly different from the results shown in Fig. 8. This difference is caused by the following reasons. First, in the cloud environment, the Yarn workload and the scientific computing applications share resources, which results in an increased execution time. Second, the FGM execution time increment is less than those of the other algorithms because the FGM adapts to the resource sharing environment and takes effective measures to avoid resource contention.

The average completion times of jobs and the distributions of job completion times are shown in Figs. 15 and 16. Figure 15 shows that the improvements achieved by the FGM compared with the other algorithms are between 38.25 and 63.36%. This result largely occurs because the scientific computing applications increase the job execution times and because the increment of the FGM is less than those of the other algorithms. Figure 16 shows that the maximum job completion time is approximately 800 s, which is far better than the results for other algorithms.

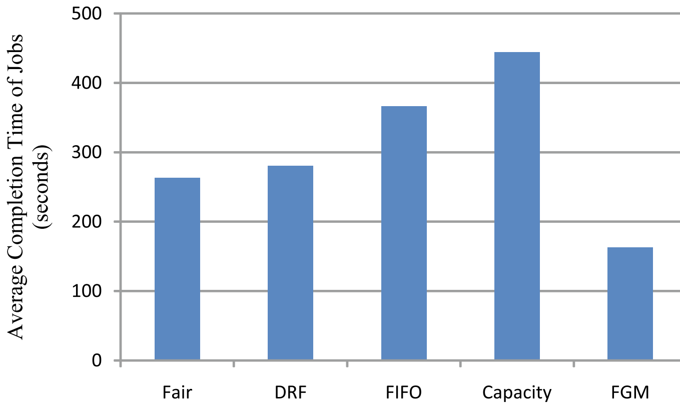

Fig. 15 Average completion time of jobs in the cloud environment using the online workload

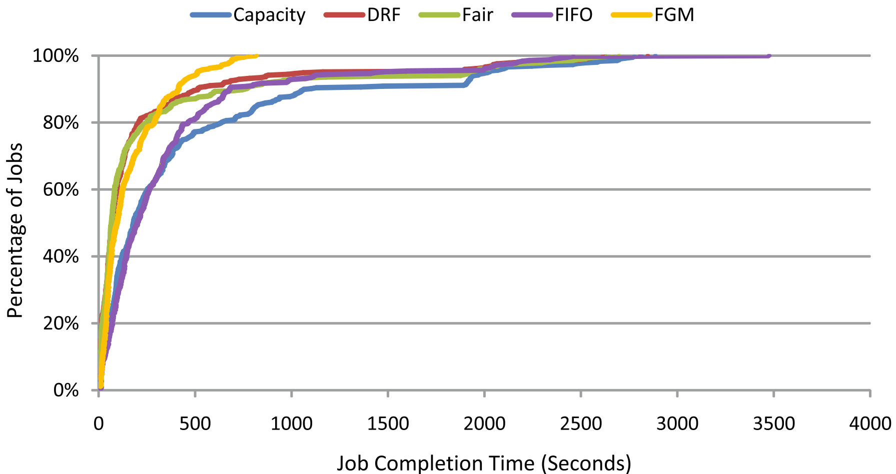

Fig. 16 CDF of job completion times in the cloud environment using the online workload

The completion times of online workloads in the cloud environment are shown in Fig. 17. Consistent with the results obtained in the dedicated environment, the FGM achieves better performance than other algorithms, and the improvement increases with workload data size. The difference in the results of the two test environments is that the workload completion time in the cloud environment is greater than the workload completion time in the dedicated environment when the workload data size is the same. The maximum performance improvement achieved by the FGM is approximately 65%, which is greater than the test results in the dedicated environment. Three factors contribute to this situation. First, the FGM improves resource utilization using several mechanisms such as fine-grained matching and compression allocation. Second, the FGM avoids resource contention by analyzing server resource availability and adjusting the server scheduling policy. This step softens the impact of resource sharing on the execution environment. In addition, the other algorithms are influenced by the

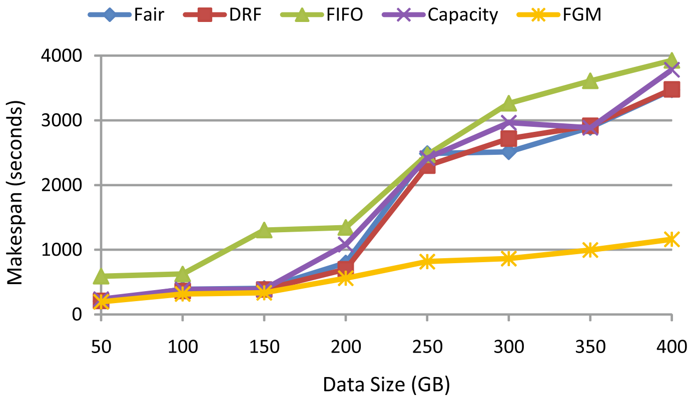

Fig. 17 Completion times of online workloads in the cloud environment

Table 2 Completion rate of input data in the resource contention environment

Fair	DRF	FIFO	Capacity	FGM
98.44%	98.06%	85.4%	91.28%	99.85%

scientific computing applications in the cloud environment. This condition leads to a sharp increase in workload completion time and highlights the optimization achieved by the FGM. Based on the test results in Fig. 17, it can conclude that the FGM significantly improves resource utilization and performance of the cloud platform in the cloud environment.

5.5 Test in the resource contention environment with the online workload

The data completion rates achieved by the different algorithms in resource contention environment are shown in Table 2. The results show that the FGM completion rate is the highest and the FIFO completion rate is lowest. The difference between the two algorithms is 14.45%. The FGM processes 115.6 GB more data than FIFO in the resource contention environment when the total data size is 800 GB. This improvement occurs because the FGM is aware of and avoids resource contention. None of the algorithms in the table process all data. This result may be caused by the burst scientific computing applications. These applications cause serious resource contention and may lead to unexpected crashes of the DataNode service in a distributed file system, resulting in a missing data block.

6 Conclusion

This paper proposes a fine-grained method called FGM for resolving resource fragmentation and over-commitment problems. This method estimates resource information at runtime and divides tasks into execution stages according to actual requirements. Then, the FGM matches the resource requirement of the task and

available server resources by stages in order to refine allocation granularity and avoid resource fragmentation and over-commitment. FGM innovatively introduces controllable compression into resource matching and allocation. The FGM can compress resource allocation to further improve resource utilization and performance. In addition, this method uses several mechanisms such as runtime resource monitoring, allocation policy adjustment and compression rate limitation to ensure efficient resource utilization and performance. The paper performed experiments in three environments using both online and offline workloads. The test results showed that the fine-grained method can improve performance in the dedicated environment by 30% and in the cloud environment by 65%. Thus, the FGM resolves resource fragmentation and over-commitment problems, and improves resource utilization and performance with acceptable fairness and scheduling response times.

FGM is suitable for the workload in which resource requirements are diverse and dynamic. In this case, FGM can improve matching degree between resource supply and demand. But when the resource requirements are similar and fixed, the computing cost of FGM is larger than other simple algorithms. And FGM is not suitable for the workload which is dominated by small jobs considering cost of matching and resource compression. Furthermore, FGM may be limited by computing capacity of server when the cluster scale is very large. In this case, FGM can be combined with distributed scheduling architecture to meet the scale challenge.

In future work, we plan to support more mechanisms, such as load balancing. In addition, we will concentrate on improving the efficiency of the FGM.

Acknowledgements This work was supported by the National Key Research and Development Program of China (No. 2016YFB0200902 to X. Zhang); and the National Natural Science Foundation of China (No. 61572394 to X. Dong).

References

1. Reiss C, Tumanov A, Ganger GR, Katz RH, Kozuch MA (2012) Heterogeneity and dynamicity of clouds at scale: Google trace analysis. In: Proceedings of the 3rd ACM Symposium on Cloud Computing, pp 7–19
2. Staples G (2006) TORQUE resource manager. In: Proceedings of the 2006 ACM/IEEE Conference on Supercomputing
3. Capacity Scheduler. https://hadoop.apache.org/docs/r2.6.0/hadoop-yarn/hadoop-yarn-site/CapacityScheduler.html. Accessed 14 July 2017
4. Zaharia M, Borthakur D, Sarma JS, Elmeleegy K, Shenker S, Stoica I (2009) Job scheduling for multi-user MapReduce clusters. EECS Department, University of California, Berkeley, Technical report UCB/EECS-2009-55
5. Zaharia M, Borthakur D, Sen Sarma J, Elmeleegy K, Shenker S, Stoica I (2010) Delay scheduling: a simple technique for achieving locality and fairness in cluster scheduling. In: Proceedings of the 5th ACM European Conference on Computer Systems, pp 265–278
6. Zaharia M, Konwinski A, Joseph AD, Katz RH, Stoica I (2008) Improving MapReduce performance in heterogeneous environments. In: Proceedings of the 8th Symposium on Operating Systems Design and Implementation, vol 8, pp 29–42
7. Apache Hadoop. http://hadoop.apache.org/. Accessed 14 July 2017
8. Ousterhout K, Wendell P, Zaharia M, Stoica I (2013) Sparrow: distributed, low latency scheduling. In: Proceedings of the 24th ACM Symposium on Operating Systems Principles, pp 69–84

9. Reiss C, Tumanov A, Ganger GR, Katz RH, Kozuch MA (2012) Towards understanding heterogeneous clouds at scale: Google trace analysis. Intel Science and Technology Center for Cloud Computing, Technical report ISTC-CC-TR-12-101
10. Abdul-Rahman OA, Aida K (2014) Towards understanding the usage behavior of Google cloud users: the mice and elephants phenomenon. In: Proceedings of the 6th IEEE International Conference on Cloud Computing Technology and Science, pp 272–277
11. Di S, Kondo D, Cappello F (2013) Characterizing cloud applications on a Google data center. In: Proceedings of the 42th IEEE International Conference on Parallel Processing, pp 468–473
12. Boutin E, Ekanayake J, Lin W, Shi B, Zhou J, Qian Z, Wu M, Zhou L (2014) Apollo: scalable and coordinated scheduling for cloud-scale computing. In: Proceedings of the 11th USENIX Symposium on Operating Systems Design and Implementation, pp 285–300
13. Schwarzkopf M, Konwinski A, Abd-El-Malek M, Wilkes J (2013) Omega: exible, scalable schedulers for large compute clusters. In: Proceedings of the 8th ACM European Conference on Computer Systems, pp 351–364
14. Grandl R, Ananthanarayanan G, Kandula S, Rao S, Akella A (2014) Multi-resource packing for cluster schedulers. In: Proceedings of the ACM Conference on SIGCOMM, pp 455–466
15. Lu P, Lee YC, Wang C, Zhou BB, Chen J, Zomaya AY (2012) Workload characteristic oriented scheduler for MapReduce. In: Proceedings of the 18th IEEE International Conference on Parallel and Distributed Systems, pp 156–163
16. Tian C, Zhou H, He Y, Zha L (2009) A dynamic MapReduce scheduler for heterogeneous workloads. In: Proceedings of the 8th IEEE International Conference on Grid and Cooperative Computing, pp 218–224
17. Tang Z, Liu M, Ammar A, Li K, Li K (2016) An optimized MapReduce work ow scheduling algorithm for heterogeneous computing. J Supercomput 72(6):2059–2079
18. Dean J, Barroso LA (2013) The tail at scale. Commun ACM 56(2):74–80
19. Garraghan P, Ouyang X, Yang R, McKee D, Xu J (2018) Straggler root-cause and impact analysis for massive-scale virtualized cloud datacenters. IEEE Trans Serv Comput. https://doi.org/10.1109/TSC.2016.2611578
20. Ghodsi A, Zaharia M, Hindman B, Konwinski A, Shenker S, Stoica I (2011) Dominant resource fairness: fair allocation of multiple resource types. In: Proceedings of the 8th USENIX Symposium on Networked Systems Design and Implementation, pp 232–336
21. Grandl R, Chowdhury M, Akella A, Ananthanarayanan G (2016) Altruistic scheduling in multi-resource clusters. In: Proceedings of the 12th USENIX Symposium on Operating Systems Design and Implementation, pp 65–80
22. Vavilapalli VK et al (2013) Apache Hadoop YARN: yet another resource negotiator. In: Proceedings of the 4th ACM Symposium on Cloud Computing, pp 1–16
23. Zhang Z, Li C, Tao Y, Yang R, Tang H, Xu J (2014) Fuxi: a fault-tolerant resource management and job scheduling system at internet scale. Proc VLDB Endow 7(13):1393–1404
24. Verma A, Pedrosa L, Korupolu M, Oppenheimer D, Tune E, Wilkes J (2015) Large-scale cluster management at Google with Borg. In: Proceedings of the 10th ACM European Conference on Computer Systems, pp 1–17
25. Jain R, Chiu DM, Hawe WR (1984) A quantitative measure of fairness and discrimination for resource allocation in shared computer system. Technical report DEC-TR-301
26. Dean J, Ghemawat S (2008) MapReduce: simplified data processing on large clusters. Commun ACM 51(1):107–113
27. Hindman B, Konwinski A, Zaharia M, Ghodsi A, Joseph AD, Katz R, Shenker S, Stoica I (2011) Mesos: a platform for fine-grained resource sharing in the data center. In: Proceedings of the 8th USENIX Symposium on Networked Systems Design and Implementation, pp 295–308
28. Isard M, Prabhakaran V, Currey J, Wieder U, Talwar K, Goldberg A (2009) Quincy: fair scheduling for distributed computing clusters. In: Proceedings of the 22nd ACM SIGOPS Symposium on Operating Systems Principles, pp 261–276
29. Gog I, Schwarzkopf M, Gleave A, Watson RN, Hand S (2016) Firmament: fast, centralized cluster scheduling at scale. In: Proceedings of 12th USENIX Symposium on Operating Systems Design and Implementation, pp 99–115
30. Ananthanarayanan G, Kandula S, Greenberg A, Stoica I, Lu Y, Saha B, Harris E (2010) Reining in the outliers in map-reduce clusters using Mantri. In: Proceedings of the 9th USENIX Symposium on Operating Systems Design and Implementation, pp 265–278

31. Kung HT, Robinson JT (1981) On optimistic methods for concurrency control. ACM Trans Database Syst 6(2):213–226
32. Ghodsi A, Zaharia M, Shenker S, Stoica I (2013) Choosy: max–min fair sharing for datacenter jobs with constraints. In: Proceedings of the 8th ACM European Conference on Computer Systems, pp 365–378
33. Lee YH, Huang KC, Shieh MR, Lai KC (2017) Distributed resource allocation in federated clouds. J Supercomput 73(7):3196–3211
34. AlEbrahim S, Ahmad I (2017) Task scheduling for heterogeneous computing systems. J Supercomput 73(6):2313–2338
35. Agarwal S, Kandula S, Bruno N, Wu MC, Stoica I, Zhou J (2012) Re-optimizing data-parallel computing. In: Proceedings of the 9th USENIX Symposium on Networked Systems Design and Implementation, pp 281–294
36. Ferguson AD, Bodik P, Kandula S, Boutin E, Fonseca R (2012) Jockey: guaranteed job latency in data parallel clusters. In: Proceedings of the 7th ACM European Conference on Computer Systems, pp 99–112
37. Khan M, Jin Y, Li M, Xiang Y, Jiang C (2016) Hadoop performance modeling for job estimation and resource provisioning. IEEE Trans Parallel Distrib Syst 27(2):441–454
38. Ananthanarayanan G, Ghodsi A, Wang A, Borthakur D, Kandula S, Shenker S, Stoica I (2012) Pacman: coordinated memory caching for parallel jobs. In: Proceedings of the 9th USENIX Symposium on Networked Systems Design and Implementation, pp 267–280
39. Morton K, Balazinska M, Grossman D (2010) ParaTimer: a progress indicator for MapReduce DAGs. In: Proceedings of the 2010 ACM SIGMOD International Conference on Management of Data, pp 507–518
40. Zhang X, Tune E, Hagmann R, Jnagal R, Gokhale V, Wilkes J (2013) CPI2: CPU performance isolation for shared compute clusters. In: Proceedings of the 8th ACM European Conference on Computer Systems, pp 379–391