Fairness-Efficiency Scheduling for Cloud Computing with Soft Fairness Guarantees

Shanjiang Tang, Ce Yu, Yusen Li

Abstract—Fairness and efficiency are two important metrics for users in modern data center computing system. Due to the heterogeneous resource demands of CPU, memory, and network I/O for users' tasks, it cannot achieve the strict 100% fairness and the maximum efficiency at the same time. Existing fairness-efficiency schedulers (e.g., Tetris) can balance such a tradeoff elastically by relaxing fairness constraint for improved efficiency using the knob. However, their approaches are *unaware* of fairness degradation under different knob configurations, which makes several drawbacks. First, it cannot tell how much *relaxed* fairness can be guaranteed given a knob value. Second, it fails to meet several essential properties such as sharing incentive. To address these issues, we propose a new fairness-efficiency scheduler, *QKnober*, to balance the fairness and efficiency elastically and flexibly using a tunable fairness knob. QKnober is a *fairness-sensitive* scheduler that can maximize the system efficiency while guaranteeing the θ-soft fairness by modeling the whole allocation as a combination of *fairness-oriented* allocation and *efficiency-oriented* allocation. Moreover, QKnober satisfies fairness properties of sharing incentive, envy-freeness and pareto efficiency given a proper knob value. We have implemented QKnober in YARN and evaluated it using both testbed and simulated experiments. The results show that QKnober outperforms its alternatives DRF and Tetris by 31.2% and 4.5%, respectively.

Index Terms—Multi-Resource Allocation, Fairness, Efficiency, Hadoop

1 Introduction

In the current era of 'big data', it has become typical to take existing large-scale data computing frameworks such as MapReduce [13] and Spark [39] for big data analytics in a data center cluster consisting of many machines. At any time, there are many users running their data-parallel applications on the cluster. The jobs submitted by users normally contain many tasks and the tasks often have *heterogeneous* resource requirements towards different resource types (e.g., CPU and memory) [26]. In addition to the heterogeneity, current data computing platforms are often underutilized. Reiss et al. conducted a trace analysis of 12,000-server Google cluster, showing that its average memory usage is lower than 35% and the CPU utilization does not exceed 40% [25]. Similarly, Delimitrou et al. showed that the majority of machines at Twitter are below 20% utilization [14]. Moreover, an experimental study estimated that the average server utilization in Amazon EC2 is about 7.3%.

To address such inefficiency, resource sharing is an efficient and widely used approach. It is based on the following facts that 1). the tasks demands of different users are generally different and diverse with respect to different resource types; 2). even for a single user, the resource demands of its workloads are changing over time. By having multiple users to execute their workloads simultaneously in a shared cluster, resource sharing can improve the system efficiency since overloaded users can leverage the unused resources from underloaded users.

- *S.J. Tang, C. Yu are with the College of Intelligence and Computing, Tianjin University, Tianjin 300072, China.*
  E-mail: {tashj, yuce}@tju.edu.cn.
  (Corresponding authors: Shanjiang Tang and Ce Yu.)
- *Yusen Lee is with the School of Computing, Nankai University, Tianjin 300071, China.*
  E-mail: liyusen@nbjl.nankai.edu.cn.

Apart from the system efficiency, fairness is another important criteria for users in the shared cluster. Being aware of heterogeneous resource demands of users' tasks, there is a need to consider multi-resource fairness that takes multiple resource types into account. By leveraging the game-theoretic definition, a *robust* multi-resource fair allocation is the one in which

- *all users in the shared system should perform no worse than that under an exclusively non-sharing partition of the system. (Sharing incentive)*
- *no user envies the allocations of any other users. (Envy freeness)*
- *no user can increase its resource allocation without harming at least one other user. (Pareto efficiency)*

Dominant Resource Fairness (DRF) is one of the most well-known multi-resource fair allocation policies [18] with the above three game-theoretic properties. The dominant resource refers to as the resource that is heavily used by a user. The fairness is achieved by equalizing the share of each user's dominant resource. Although there have been a number of extensions [23], [24], [35], they draw little attention to the influence on system efficiency. Recent studies have shown that there is a tradeoff between fairness and efficiency in multi-resource allocation [19], [21], [34]. Guaranteeing the strict 100% fairness across users would produce inefficient resource allocations. Conversely, seeking for high system efficiency is often at the cost of compromised fairness. DRF and its extensions tend to over constrain the system for high fairness guarantee, resulting in resource allocations with low system efficiency.

Many existing fairness-efficiency schedulers seek to relax fairness (i.e., allowing some degree of unfairness) for efficiency improvement by employing knob-based heuristic algorithms [11], [19], [28], [37]. Tetris [19] is the state-of-the-art knob-based tradeoff scheduler that allows users to balance

fairness and efficiency flexibly by tuning the fairness knob in a data center cluster. However, due to its *insensitiveness* of fairness degradation under different knob configurations, there are some shortcomings (See Section 4): 1). it cannot show users how much *relaxed* fairness can be guaranteed given a fairness knob; 2). it fails to satisfy several fairness properties such as sharing incentive.

In this paper, we develop a new fairness-efficiency scheduler, *QKnober*, to allow users to balance fairness and efficiency flexibly with a knob factor $\rho \in [0, 1]$. Unlike the previous schedulers [11], [19], [37], QKnober is a *fairness-sensitive* scheduler that works on the *relaxed* fairness (i.e., *soft* fairness in Section 5.1), which refers to the maximum difference between the normalized shares of any two users. It is achieved by modeling the multi-resource allocation as a combination of *fairness-oriented* allocation and *efficiency-oriented* allocation (Section 5.1). Given a knob ρ, QKnober first performs the fairness-oriented allocation for the θ-soft fairness guarantee (See Theorem 1 in Section 5.1) and then does the efficiency-oriented allocation for maximizing the system efficiency. We show that with a proper knob configuration, QKnober can ensure that each user in the shared system can get at least the amount of resources as that under the exclusively non-sharing partition of the system. It also can guarantee that every user prefers to its own allocation and no user envies the allocations of any other users. Furthermore, QKnober keeps that the system is fully utilized by ensuring that no user can get more resource allocation without decreasing the allocation of at least one user.

We have implemented QKnober in YARN [32]. We evaluated QKnober with testbed workloads in a Amazon EC2 cluster consisting of 60 nodes. Our results show that QKnober strikes a flexible balance between fairness and efficiency. There can be up to 57% performance improvement as we decrease the knob factor from one to zero for QKnober. Moreover, it outperforms its alternatives DRF and Tetris by 31.2% and 4.5% on average, respectively. Finally, we show that the scheduling overhead of QKnober is minor (< 0.42 ms). In addition, we also conduct a simulation-based experiment at a large scale with Google cluster-usage traces. The simulation results are consistent with that of testbed experiments.

Organization. Section 3 gives the formal definition of several desirable fairness properties. Section 4 overviews existing schedulers and motivates our work by showing their shortcomings. We introduce and analyze our approach in Section 5. The experimental evaluations are presented in Section 7. We review the related work in Section 2. Finally, we conclude the paper in Section 8.

2 Related Work

We review the literature work that are close to our work from the following perspectives:

Multi-Resource Fairness. In multi-resource allocation, DRF is one of the most popular fair policies [18]. It offers fair allocation of multiple resources based on dominant resource shares. The popularity of DRF lies in its good properties including sharing incentive, envy-freeness, and pareto-efficiency. It has been implemented in many current datacenter framework, including YARN [32] and Mesos [20]. After that, there have been a lot of extension and generalization for DRF. Bhattacharya et al. [8] generalized DRF to support hierarchical scheduling. Ghodsi et al. [17] extended DRF to fair queueing of package processing. Kash et al. [22] extended the DRF model to a dynamic setting where users can join the system over time but will never leave. Wang et al. [35] generalized DRF in a distributed system with heterogeneous servers, followed by a TSF fairness policy for the case when there is a placement constraint for task placement [36]. Dolev et al. [15] generalized DRF to consider multiple contended resources by proposing bottleneck-based fairness, rather than the single dominant (bottleneck) resource only for each user. Parkes et al. [24] extended DRF in several ways and focused on in particular the case of indivisible tasks. Liu et al. [23] proposed a Reciprocal Resource Fairness by extending DRF to allow the trade among different types of resources between users. When the resource demand vector required by DRF is not available in computer architectures, Zahedi et al. [40] proposed an alternative multi-resource policy based on Cobb-Douglas utility function for multiprocessors. In summary, all the works above focus on the 100% fairness. In contrast, this paper considers tradeoff allocation between fairness and performance simultaneously.

Fairness-efficiency Scheduling. There is a general tradeoff between fairness and efficiency in multi-resource allocation, which has been studied by a lot of research works. Joe-Wong et al. [21] proposed a unifying *mathematical* framework to capture the tradeoff between fairness and efficiency, which are specified by two parameters for a given multi-resource allocation problem. However, their work is purely a theoretic study and meanwhile cannot allow users to balance the fairness and efficiency flexibly. In contrast, our proposed knob-based policy QKnober is practical and elastic. We have implemented it in Hadoop that allows users to balance the fairness-efficiency tradeoff flexibly by tuning the knob in the range of $[0, 1]$. Wang et al. [34], [37] and Danna et al. [11] studied the fairness-efficiency tradeoff for packet processing consuming both CPU and link bandwidth by proposing a GPS-like fluid model. Wang et al. [33] proposed a bottleneck-aware allocation policy to balance fairness and efficiency for users in multi-tiered storage consisting of SSD and HDD. ElasticSEM [27] is a fairness-efficiency scheduler for semi-external memory caching system consisting of memory and SSD. In contrast, we consider the big data job scheduling in the data center cluster.

EMRF [28] and Tetris [19] are two closely related work to our work. For the EMRF [28], it focuses on fairness-efficiency scheduling for Coupled CPU-GPU architecture by modeling CPU and GPU as different typed resources but considering the fairness of overall GFLOPS from CPU and GPU across users. In contrast, QKnober focuses on the fairness-efficiency job scheduling for data center scenario where CPU and memory resources are considered. Moreover, EMRF policy only considered a single machine with one APU chip, whereas QKnober works for a data center consisting of multiple machines. Tetris is a fairness-efficiency scheduler for cloud computing that balances the performance and fairness by leveraging alignment heuristics to efficiently pack tasks with heterogeneous resource demands to servers. However, it cannot provide us a soft fairness guarantee given a knob setting due to its *unawareness* of fairness

degradation (Section 4) during its fairness-efficiency scheduling. Moreover, it doesn't satisfy sharing incentive property. In comparison, our proposed knob-based policy QKnober is fairness-sensitive, which maximizes the efficiency while guaranteeing the θ-soft fairness under a knob configuration (See Theorem 1). Additionally, it satisfies sharing incentive, envy freeness and pareto efficiency properties.

3 Desirable Allocation Properties

From the economic point of view, a *good* fair allocation policy in cluster computing system should provide the following essential game theoretic properties, including sharing incentive, envy-freeness, and pareto efficiency.

Sharing Incentive (SI): Resource sharing is an essential and effective approach to improve the system utilization and efficiency. A good allocation policy should satisfy sharing incentive (SI) such that each user in the system performs at least as good as it would be under a statically equal split of the resources of the computing system. Otherwise, users would not be willing to share their resources with others. Thus, to enable resource sharing possible and sustainable, it is a must requirement to satisfy sharing incentive.

Formally, let $\mathbf{U}_i = \langle u_{i,1}, \cdots, u_{i,m} \rangle$ be the resource allocation vector for user i. Let $N_i(\mathbf{U}_i)$ denote the number of tasks scheduled for user i under the resource allocation vector $\mathbf{U}_i$. An allocation policy is sharing incentive if it satisfies the following condition for each user $i \in [1, n]$,

$$N_i(\mathbf{U}_i) \geqslant N_i(\overline{\mathbf{U}}_i), \tag{1}$$

where $\overline{\mathbf{U}}_i = \langle \overline{u}_{i,1}, \cdots, \overline{u}_{i,m} \rangle$ represents the resource allocation vector for user i under the exclusively non-sharing partition of the computing system.

Envy-freeness (EF): An allocation is envy-freeness (EF) if no user envies the allocation of other users associated with a desire to receive that same allocation. That is, every user prefers its own allocation to that of any other user. To provide EF, there is a need to ensure that every user cannot have more tasks scheduled by switching its allocation with any other user.

Given the resource allocation vector $\mathbf{U}_i$ for user i, an allocation policy satisfies EF if

$$N_i(\mathbf{U}_i) \geqslant N_i(\mathbf{U}_j), \tag{2}$$

for any two users $i, j \in [1, n]$.

Pareto Efficiency (PE): PE is another critical property that should be satisfied by a fair resource allocation policy [31]. It is essential for high resource utilization and efficiency. An allocation policy is PE if it is not possible for a user to get more tasks scheduled without decreasing the number of running tasks of at least one other user.

Let $\mathbf{U} = \langle \mathbf{U}_1, \cdots, \mathbf{U}_n \rangle$ be the resulting allocation for all users produced by a fair allocation policy. The allocation $\mathbf{U}$ is PE if it does not exist any feasible allocation $\breve{\mathbf{U}}$ satisfying the following two conditions at the same time, i.e., 1). $\forall i \in [1, n], N_i(\mathbf{U}_i) \leqslant N_i(\breve{\mathbf{U}}_i)$; 2). $\exists j \in [1, n], N_j(\mathbf{U}_j) < N_j(\breve{\mathbf{U}}_j)$.

4 Background and Motivation

In this section, we motivate our work by reviewing and analyzing the limitations of existing schedulers.

***Fairness* vs. *System Efficiency*.** For the *single* resource allocation, we can achieve fairness across multiple users by dividing the system resources in the ratio of their assigned weights. For example, if there is a computing system with 100 CPUs shared by two users equally, then each user receives 50 CPUs. Many slot-based analytic frameworks like Hadoop [32] and Spark [39] support the single resource allocation, which define slots based on one resource (e.g., CPU). Their fair schedulers are work-conserving schedulers, which can make the system fully utilized as long as there are pending tasks. It indicates that we can achieve both *strict* fairness and high system efficiency at the same time for the single-resource allocation.

However, when it comes to the multi-resource allocation, the situation becomes more complicated and challenging. The fairness and system utilization/efficiency highly depend on the workload characteristic and allocation ratio of the resource. If the users with memory-intensive workloads have small allocation ratio, it may result in low utilization for the memory resource due to insufficient requests. Similar case does also hold for other resources (e.g., CPU). On the contrary, maintaining high resource utilization for all resources often generate the allocations in a manner that starves some users, resulting in unfairness problem for users in the allocation. We next demonstrate these problems using examples of Dominant Resource Fairness (DRF) [18], which is a popular multi-resource allocation policy with many attractive merits (e.g., sharing incentive, envy-freeness and pareto-efficiency).

Example 1: Consider a computing system made up of 200 CPUs and 1000 GB memory in total. It is shared by two users A and B equally with the task requirement of $\langle$1 CPU, 6 GB $\rangle$ for A and $\langle$1 CPU, 2 GB $\rangle$ for B, respectively.

The dominant resource for user A is memory because each task of A consumes $1/200$ of the total CPUs and $6/1000$ of the total memory, while the dominant resource for B is CPU. DRF achieves the fairness by equalizing the dominant resource shares (i.e., $546/1000 = 109/200$) for A and B, with the resulting allocation illustrated in Figure 1. The memory utilization is only $\frac{(546+218)}{1000} \approx 76\%$. This is because DRF does not consider resource efficiency when making allocation decision. It only focuses on achieving the fairness among users, but does not deal with the impact of such adjustments on the system efficiency.

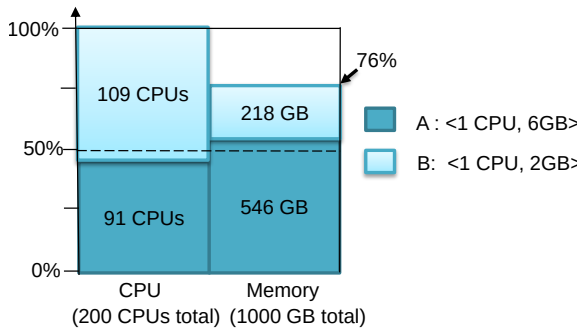


Fig. 1: Allocation with DRF for Example 1. The utilization of memory is only 76%.

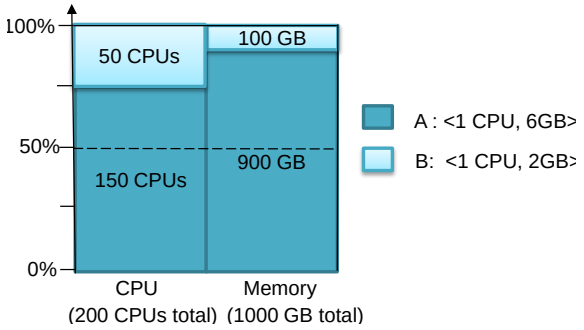


Fig. 2: Allocation with 100% utilization for both CPU and memory of Example 1.

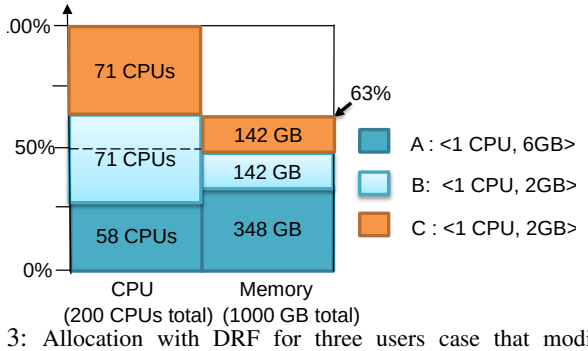


Fig. 3: Allocation with DRF for three users case that modifying Example 1 by adding a third user C of $\langle 1$ CPU, 2 GB$\rangle$ per task. The memory utilization decreases from 76% of Example 1 to 63%.

In fact, both CPU and memory resources in Example 1 can be fully utilized if the scheduler allocates $\langle 150$ CPUs, 900 GB$\rangle$ to A and $\langle 50$ CPUs, 100 GB$\rangle$ to B, as illustrated in Figure 2. However, the dominant resource shares of A and B are no longer the same (i.e., $\frac{900}{1000} > \frac{50}{200}$), being *unfair* for B. It implies that there tends to be a tradeoff between fairness and system efficiency in resource allocation.

Moreover, the system utilization is also highly dependent on the competing workloads and can be worsen under DRF policy with more users joining in the system. To demonstrate this, we modify Example 1 by adding a third user C also with resource demand of $\langle 1$ CPU, 2 GB$\rangle$ per task. Figure 3 illustrates the allocation of DRF policy. It achieves the fairness by allocating $\langle 58$ CPUs, 348 GB$\rangle$, $\langle 71$ CPUs, 142 GB$\rangle$ and $\langle 71$ CPUs, 142 GB$\rangle$ to A, B and C, respectively. The memory-intensive user B is severely throttled by DRF fairness constraint requirement, making memory utilization decrease from 76% in Example 1 (Figure 1) to 63%. (Figure 3).

Flaws of Existing Fairness-Efficiency Schedulers. To balance the tradeoff between fairness and efficiency elastically and flexibly, many fairness-efficiency schedulers [19], [28], [34], [37] take *knob-based* heuristics, which is *promised* as an effective approach in multi-resource allocation [19] and there are many different definitions and implementations for it. Wang et al. [34], [37] studied the fairness-efficiency tradeoff in networking system by considering network packet processing and data transfer flow across different machines. EMRF [28] is a fairness-efficiency tradeoff scheduler for Coupled CPU-GPU architectures. In contrast, Tetris [19] is the *state-of-the-art* knob-based scheduler for cloud computing. Specifically, in each resource allocation, it first sorts all tasks according to the DRF. Then, it searches the best task for efficiency among the runnable tasks belonging to the first $(1-f)$ tasks in the sorted list, where $f \in [0, 1]$ is a knob provided by users in advance. It computes the *alignment score*, defined as the weighted dot product between the vector of machine's available resources and the task's peak resource demand, to the machine for each task, and the best task is picked with the largest alignment score. However, there are several flaws for Tetris as follows:

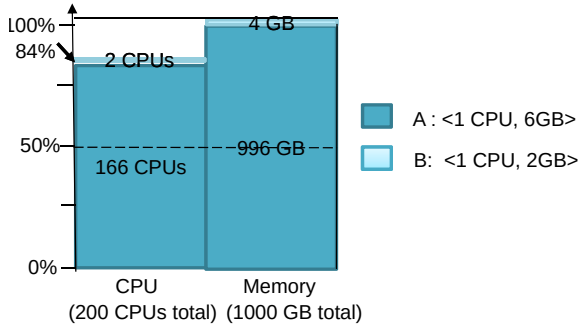


Fig. 4: The resulting allocation with Tetris for Example 1 when the knob f satisfies $0 \leqslant f \leqslant 0.45$. In this case, different value of knobs does not work for fairness improvement, indicating that Tetris is *insensitive* to fairness under different knobs.

First, although Tetris allows users to relax fairness for efficiency improvement by tuning the fairness knob, it is *insensitive* to fairness degradation for different knobs during the allocation. Particularly, it cannot show users how much *relaxed* fairness (i.e., *soft* fairness in Section 5.1) can be guaranteed given a knob configuration. To explain it, let's revisit Example 1 by assuming that there two users A and B each with one job consisting of 1000 tasks, respectively. We can see that the task (share: $\langle \frac{1}{200}, \frac{6}{1000} \rangle$) of A's job is more beneficial to the system utilization than that (share: $\langle \frac{1}{200}, \frac{2}{1000} \rangle$) of B's job according to the resource type difference of their tasks ($A : |\frac{1}{200} - \frac{6}{1000}| = \frac{1}{1000}$, $B : |\frac{1}{200} - \frac{2}{1000}| = \frac{3}{1000}$). In this case, Tetris first sorts all 2000 tasks from A's job and B's job according to DRF policy. Next it tries to pick up a task from the first $(1-f)$ tasks in the sorted list that is most beneficial to the system efficiency. When $0 \leqslant f \leqslant 0.45$, the task range that Tetris can choose at the second stage is $1100 < (1-f) * 2000 \leqslant 2000$. Then Tetris always picks up preferred tasks from A's job rather than B's job until it cannot fulfilled, resulting in allocation as shown in Figure 4 for all $0 \leqslant f \leqslant 0.45$. It shows that the knob of Tetris does not work for fairness improvement when $0 \leqslant f \leqslant 0.45$, i.e., Tetris is *insensitive* to fairness degradation under different knobs, implying that it cannot tell users how much relaxed fairness can be guaranteed under a knob setting. However, in practice, the guarantee of different levels of fairness is very important for users under different knobs configurations.

Third, Tetris violates the sharing incentive property (See definition in Section 3). Let's take Example 1 as a counterexample to demonstrate it. Provided that $f = 0$, Tetris is purely for system efficiency optimization by picking the best task for efficiency among all tasks every time, resulting in the allocation as illustrated in Figure 2. We can see that, B receives less resources (i.e., fewer tasks scheduled) in the sharing system than that (i.e., $\langle 100$ CPUs, 200 GB$\rangle$) of exclusively using its partition

of the system without sharing, violating the sharing incentive property.

Motivated by these, we seek to explore a new fairness-efficiency allocation policy that guarantees the soft fairness and satisfies all the desirable properties listed in Section 3.

5 Allocation Model and Scheduling Policy

In this section, we model the multi-resource allocation in cloud computing based on DRF, and then propose a fairness-efficiency scheduling policy called QKnober.

5.1 Multi-Resource Allocation Model

Basic Setting. We start by defining some terms used in our model. Suppose that the computing system consists of m resource types (e.g., CPU, memory, disk) with the capacity of $\mathbf{R} = \langle r_1, \cdots, r_m \rangle$ shared by n users, where r_i denotes the total amount of resource i. For each user i, let w_i denote its share weight in the shared computing system and $\mathbf{D}_i = \langle d_{i,1}, \cdots, d_{i,m} \rangle$ be its *resource demand vector*, where $d_{i,j}$ denotes the amount of resource j required by a task of user i. We assume that each user has an infinite number of tasks to be scheduled, and all its tasks are divisible and with the same resource demand. We later discuss how these assumptions can be relaxed for practical usage in Section 6.1.

Given the allocation matrix $\mathbf{U} = \langle \mathbf{U}_1, \cdots, \mathbf{U}_n \rangle$ for all users, it is a *feasible* allocation if it satisfies that,

$$\sum_{i=1}^{n} u_{i,k} \leqslant r_k, \quad \forall k \in [1, m]. \tag{3}$$

The maximum number of tasks $N_i(\mathbf{U}_i)$ (possible fractional) that user i can schedule under the resource allocation vector $\mathbf{U}_i$ is,

$$N_i(\mathbf{U}_i) = \min_{1 \leqslant k \leqslant m} \{u_{i,k}/d_{i,k}\}. \tag{4}$$

Allocation Model. An efficient resource allocation should never let a user get more resources than it actually needs in the computing system. We call such an allocation *non-wasteful*. Formally, an allocation $\mathbf{U}_i$ is *non-wasteful* if and only if it satisfies the following condition:

$$\mathbf{U}_i = N_i(\mathbf{U}_i) \cdot \mathbf{D}_i. \tag{5}$$

It is worthy mentioning that we can always convert an allocation to the *non-wasteful* allocation by transferring the redundant/unused resources of each user to other potential users without decreasing the number of tasks scheduled for that user. Without loss of generality, in the following discussions, we limit our focus on the non-wasteful allocation.

Scheduling tasks to the computing system is analogous to the multi-dimensional knapsack problem [10] by viewing the computing system as a knapsack and each task as a knapsack item. The weight of an item (or task) from user i is $\mathbf{D}_i$. In this work, since we are interested in the efficiency of resource allocation, the value of an item (or task) is the sum of the amount of different typed resources it required (normalized to the system capacity), i.e., $\sum_{k=1}^{m} d_{i,k}/r_k$. Let $\epsilon_i(\mathbf{U}_i)$ be the efficiency (i.e, knapsack cost value) of a feasible resource allocation $\mathbf{U}_i$ contributed by user i in the system. According to the knapsack problem, we have

$$\epsilon_i(\mathbf{U}_i) = N_i(\mathbf{U}_i) \cdot \sum_{k=1}^{m} d_{i,k}/r_k, \tag{6}$$

for a single user i. Then the efficiency $\epsilon(\mathbf{U})$ of a feasible allocation $\mathbf{U}$ for all users in the system can be calculated as

$$\epsilon(\mathbf{U}) = \sum_{i=1}^{n} \epsilon_i(\mathbf{U}_i) = \sum_{i=1}^{n} \{N_i(\mathbf{U}_i) \cdot \sum_{k=1}^{m} d_{i,k}/r_k\}. \tag{7}$$

Let s_i denote the share of dominant resource for user i in the computing system. According to Formula (5), we have

$$s_i = \max_{1 \leqslant k \leqslant m} u_{i,k}/r_k = N_i(\mathbf{U}_i) \cdot \max_{1 \leqslant k \leqslant m} d_{i,k}/r_k. \tag{8}$$

Formula (8) indicates that there is a proportional relationship between a user's dominant resource share and the number of tasks scheduled. The Dominant Resource Fairness (DRF) achieves the fairness by guaranteeing that the (weighted) shares of dominant resource across users are the same, i.e.,

$$\frac{s_1}{w_1} = \frac{s_2}{w_2} = \cdots = \frac{s_n}{w_n}. \tag{9}$$

Let s_i^{max} and $N_i(\mathbf{U}_i^{max})$ represent the maximum share of dominant resource and the corresponding number of tasks scheduled for user i under the DRF allocation. The DRF allocation can be viewed as progressive filling when all tasks are divisible [18]. The allocation terminates when at least one typed resource is fulfilled. In that case, we are unable to increase each user's dominant resource. That is, the dominant resource share and the corresponding number of tasks scheduled are maximized for each user under DRF. It thus holds,

$$\max_{1 \leqslant k \leqslant m} \{\sum_{i=1}^{n} \frac{u_{i,k}}{r_k}\} = \max_{1 \leqslant k \leqslant m} \{\sum_{i=1}^{n} \frac{N_i(\mathbf{U}_i) \cdot d_{i,k}}{r_k}\} = 1. \tag{10}$$

$$\Leftrightarrow \sum_{i=1}^{n} \frac{N_i(\mathbf{U}_i) \cdot d_{i,k}}{r_k} = 1 \Leftrightarrow r_k = \sum_{i=1}^{n} N_i(\mathbf{U}_i) \cdot d_{i,k}, \quad \exists k \in [1, m].$$

By computing $N_i(\mathbf{U}_i)$ with Formula (8) (9) (10), we can derive $N_i(\mathbf{U}_i^{max})$ as follows:

$$N_i(\mathbf{U}_i^{max}) = w_i / \{\phi \cdot \max_{1 \leqslant k \leqslant m} \frac{d_{i,k}}{r_k}\},$$

where $\phi = \max_{1 \leqslant k \leqslant m} \{\frac{1}{r_k} \cdot \sum_{j=1}^{n} \frac{w_j \cdot d_{j,k}}{\max_{1 \leqslant k' \leqslant m} \{\frac{d_{j,k'}}{r_{k'}}\}}\}$.

According to Formula (8), we can get s_i^{max} as

$$s_i^{max} = w_i/\phi, \tag{11}$$

5.2 QKnober

Recall that the model in Section 5.1 is a strict 100% fairness allocation model. By altering the model slightly, we can develop a knob-based fairness-efficiency scheduler, *QKnober*, to allows users to balance fairness and system efficiency flexibly using a fairness knob.

The basic idea is as follows. Instead of strictly seeking for 100% fairness as DRF does, we can compromise fairness for increased allocation efficiency by tolerating some degree of fairness loss. Particularly, we classify the fairness into two types: *hard fairness* and *soft fairness*. The hard fairness refers to that the allocation shares of all users should be the same (i.e., Formula (9) should be guaranteed). In contrast, the soft fairness tolerates

some degree (marked by θ) of unfairness across users. Formally, we define *θ-soft fairness* by changing Formula (9) as follows:

$$|\frac{s_i}{w_i} - \frac{s_j}{w_j}| \leqslant \theta, \forall i, j \in [1, n]. \quad (12)$$

Typically, DRF focuses on the hard fairness across users, limiting the allocation efficiency improvement. In contrast, QKnober, as a fairness-efficiency tradeoff scheduling policy, is interested in the soft fairness, which can leave some room for efficiency improvement. In the following, we describe our design of QKnober policy.

5.2.1 QKnober Design

The fairness-efficiency tradeoff allocation can be considered as an integration of two stages allocations: *fairness-oriented* allocation (i.e., purely for fairness optimization) and *efficiency-oriented* allocation (i.e., purely for efficiency optimization). For QKnober, it first does the fairness-oriented allocation with DRF to achieve the soft fairness guarantee. Next it turns to the efficiency-oriented allocation for efficiency maximization. Particularly, it offers users a knob $\rho \in [0, 1]$ to control and balance the two stages allocations flexibly. Let $\bar{s_i}$ and $s_i^{'}$ be the dominant resource shares of the resulting allocation for user i in the stage of fairness-oriented allocation and efficiency-oriented allocation, respectively. By combining the allocations of two stages, we get the final dominant resource share s_i for each user i as follows:

$$s_i = \bar{s_i} + s_i^{'}. \quad (13)$$

Fairness-oriented Allocation. In the stage of fairness-oriented allocation, instead of guaranteeing the hard (dominant resource) fairness of s_i^{max} for each user i, QKnober seeks to guarantee the soft fairness of $s_i^{max} \cdot \rho$ (i.e., $\bar{s_i} = s_i^{max} \cdot \rho$). According to Formula (13), we have

$$s_i = s_i^{max} \cdot \rho + s_i^{'}. \quad (14)$$

Let $\bar{\mathbf{U}} = \langle \bar{\mathbf{U}}_1, \cdots, \bar{\mathbf{U}}_n \rangle$ denote the fairness-oriented allocation result of QKnober. According to Formula (8), it holds $\bar{s_i} = N_i(\bar{\mathbf{U}}_i) \cdot \max_{1 \leqslant k \leqslant m} d_{i,k}$. Based on the DRF policy, we can model the fairness-oriented allocation as follows,

$$\textbf{Maximize} \quad (N_1(\bar{\mathbf{U}}_1), N_2(\bar{\mathbf{U}}_2), \cdots, N_n(\bar{\mathbf{U}}_n))$$

subject to

$$\frac{N_i(\bar{\mathbf{U}}_i) \cdot \max_{1 \leqslant k \leqslant m} d_{i,k}/r_k}{w_i} = \frac{N_j(\bar{\mathbf{U}}_j) \cdot \max_{1 \leqslant k \leqslant m} d_{j,k}/r_k}{w_j}. \quad \forall i, j \in [1, n].$$

and

$$\sum_{i=1}^{n} \{d_{i,k} \cdot N_i(\bar{\mathbf{U}}_i)\} \leqslant r_k \cdot \rho, \quad \forall i \in [1, n].$$

It leaves $\mathbf{R}^{'} = \langle r_1^{'}, \cdots, r_m^{'} \rangle$ idle resources for efficiency-oriented allocation, where $r_k^{'} = r_k - \sum_{j=1}^{n} \frac{s_j^{max} \cdot \rho \cdot d_{j,k}}{\max_{1 \leqslant k' \leqslant m} d_{j,k'/r_{k'}}}$ $= r_k - \sum_{j=1}^{n} N_k(\mathbf{U}_k^{max}) \cdot \rho \cdot d_{j,k}$ according to Formula (8). The small value of ρ favors the efficiency optimization. In contrast, the large value of ρ can make the fairness-oriented allocation dominant, benefiting more for fairness optimization. Typically, when $\rho = 1$, there must exist $k \in [1, m]$ satisfying $r_k = \sum_{j=1}^{n} N_k(\mathbf{U}_k^{max}) \cdot \rho \cdot d_{j,k} \Leftrightarrow r_k^{'} = 0$ according to Formula (10). It indicate that no task can be allocated in this case and $s_i^{'} = 0$ given $\rho = 1$. According to Formula (14), QKnober reduces to DRF when $\rho = 1$.

Theorem 1: QKnober is a θ-soft fairness policy where

$$\theta = \begin{cases} \max_{1 \leqslant i \leqslant n} \{ \frac{\max_{1 \leqslant k \leqslant n} d_{i,k}/r_k}{\max_{1 \leqslant k \leqslant m} \frac{w_i \cdot d_{i,k}}{r_k - \rho \cdot \sum_{j=1}^{n} N_k(\mathbf{U}_k^{max}) \cdot d_{j,k}}} \}, & (0 \leqslant \rho < 1) \\ 0, & (\rho = 1). \end{cases}$$

The proof of Theorem 1 can be found in Appendix A of the supplemental material.

Efficiency-oriented Allocation. Theorem 1 shows that the fairness-oriented allocation of QKnober can guarantee *θ-soft fairness* across users. In the second stage, we perform the efficiency-oriented allocation with the remaining idle resource vector $\mathbf{R}^{'}$ so that its overall efficiency is maximized.

Formally, our work is to search a feasible allocation $\mathbf{U}^{'}$ such that Formula (7) is maximized. Particularly, for any two users i and j with the same normalized task demands (i.e., $\frac{\mathbf{D}_i}{|\mathbf{D}_i|} = \frac{\mathbf{D}_j}{|\mathbf{D}_j|}$), exchanging resources between them has no impact on efficiency but could affect fairness. In order to achieve better fairness, we still keep Formula (9) holding for any two users satisfying $\frac{\mathbf{D}_i}{|\mathbf{D}_i|} = \frac{\mathbf{D}_j}{|\mathbf{D}_j|}$ by adding Formula (16). We can model the efficiency-oriented allocation as a linear integer programming optimization problem as follows:

$$\textbf{Maximize} \quad \epsilon(\mathbf{U}^{'}) = \sum_{i=1}^{n} \{N_i(\mathbf{U}_i^{'}) \cdot \sum_{k=1}^{m} d_{i,k}/r_k\}. \quad (15)$$

subject to:

$$s_i^{'}/w_i = s_j^{'}/w_j. \quad (\mathbf{D}_i/|\mathbf{D}_i| = \mathbf{D}_j/|\mathbf{D}_j|, \forall i, j \in [1, n]). \quad (16)$$

and

$$\sum_{i=1}^{n} \{d_{i,k} \cdot N_i(\mathbf{U}_i^{'})\} \leqslant r_k - \sum_{j=1}^{n} \frac{s_j^{max} \cdot \rho \cdot d_{j,k}}{\max_{1 \leqslant k' \leqslant m} d_{j,k'/r_{k'}}}. \quad (17)$$

for $\forall k \in [1, m]$. By resolving the linear integer program, the optimal (maximum) value of $\epsilon(\mathbf{U}^{'})$ can be obtained. Finally, the total system efficiency $\epsilon(\mathbf{U})$ can be computed by combining the allocation efficiencies in the two allocation phases.

To summarize, we have shown that QKnober is a knob-based fairness-efficiency scheduling policy that can maximize the system efficiency while guaranteeing the *θ-soft fairness* with the provided knob ρ. Particularly, different configurations of the fairness knob ρ can result in different soft fairness guarantees for QKnober (i..e, QKnober is *sensitive* to the fairness degradation under different knobs).

5.2.2 Properties Analysis of QKnober

We give an analysis of the three essential properties defined in Section 3 for QKnober.

Theorem 2: ***(Sharing Incentive):*** The QKnober allocation policy is sharing incentive when

$$\rho \geqslant (\max_{1 \leqslant k \leqslant m} \{\frac{1}{r_k} \cdot \sum_{j=1}^{n} \frac{w_j \cdot d_{j,k}}{\max_{1 \leqslant k' \leqslant m} \{\frac{d_{j,k'}}{r_{k'}}\}}\}) / \sum_{j=1}^{n} w_j.$$

The proof of Theorem 2 is given in Appendix B of the supplemental material.

By properly configuring the knob ρ according to Theorem 2, QKnober can guarantee that each user can schedule at least as the number of tasks as that under exclusively using its own partition of the system resources with no sharing. Next, we show that

QKnober is envy-freeness, namely, no user envies the allocation results of any other users under its allocation.

Theorem 3: ***(Envy Freeness):*** Every user under the QKnober allocation prefers its own allocation to others.

The proof of Theorem 3 can be found in Appendix C of the supplemental material.

We next show that QKnober produces an efficient allocation under which no user can increase its allocation without decreasing that of other users.

Theorem 4: ***(Pareto Efficiency):*** The QKnober allocation policy is pareto efficient.

Please refer to Appendix D of the supplemental material for the proof of Theorem 4.

Moreover, we also have a discussion on the discrete resource allocation for QKnober in the Appendix E of the supplemental material.

6 Implementation of QKnober

YARN has been a de facto distributed resource management system that enables many big data processing frameworks (e.g., MapReduce [12], Spark [39], HIVE [30], HBase [16]) running on top of it to share a computing cluster efficiently. In this section, we implement QKnober policy in YARN by developing a fairness-efficiency scheduler called *QKYARN*. We start with practical considerations in real world, followed by the detailed implementation of QKnober in YARN.

6.1 Practical Considerations

In our former discussions of QKnober policy, there are several key assumptions that may not be the case in a real-world computing system. For practical application of QKnober, we need to relax these assumptions by considering complicated and challenging factors for real applications and computing system. In the following, we highlight these challenges and then give our solutions to address them in YARN.

***C1*: Online Users with a Finite Number of Tasks.** In the previous discussions, it has assumed that there are an infinite number of tasks for each user at any time. However, in practice, the tasks of users are arriving over time, implying that the number of tasks per user is generally finite at a time.

Iterative QKnober Approach. We can address this problem through a small modification on QKnober as follows. First, we classify all users into two kinds: *active users* (i.e., with pending tasks) and *inactive users* (i.e., with no pending tasks). The system maintains the list of active users, where an inactive user becomes active whenever there arrives a pending task of it. The system performs QKnober allocation iteratively. In each allocation round, the system uses progressive filling approach to allocates resources to active users based on QKnober until one of them has all its pending tasks scheduled. After that, the active user becomes inactive and will not be considered in the following allocation. The system then starts a new allocation round and repeats the above allocation procedure until there is no active user or no sufficient idle resources that can be allocated.

***C2*: Heterogeneous and Indivisible Tasks.** In QKnober allocation model, we have assumed that tasks are divisible and all the tasks of a user are homogeneous in their resource demands. However, in the real world, it may not be the case. First, the tasks demands of a user are most likely to be diverse (e.g., different demands between map and reduce tasks of a user's MapReduce job). Second, fractional tasks are often not supported and accepted by existing systems (e.g., MapReduce, Spark).

In QKnober, whether to perform fairness-oriented allocation or to do efficient-oriented allocation is determined by the maximum dominant resource share s_i^{max} and provided knob ρ (See Section 5.1). When the demands of all the tasks of a user are homogeneous, the maximum dominant resource share is fixed and can be estimated by Formula (11) for each user. However, in the heterogeneous case, it varies dynamically with the running and new arriving tasks. Moreover, estimating the maximum dominant resource share in such case is *NP-hard*.

Fitness-based Approximation Approach. We propose a heuristic approach based on the First-Fit algorithm [9]. The algorithm first estimates the *current* average resource demand of tasks based on its running and pending tasks for each user. Then, it computes the maximum dominant resource share for each user by using its current average resource demand of tasks with Formula (11). However, in practice, there could be a large number of pending tasks at runtime. It indicates that picking all pending tasks might not reflect the *current* average resource demand of a user. To address it, we instead only consider a certain number of tasks that just fill the remaining idle space of the cluster. We achieve and update it for current average resource demand by using the First-Fit algorithm dynamically. That is, we count the pending tasks in the queue order until the cluster can be filled. Then the current average resource demand can be estimated based on the running tasks and those counted pending tasks. Therefore, the time complexity for estimating the maximum dominant resource share s_i^{max} is $O(n)$.

***C3*: Heterogeneous and Distributed Computing System.** The QKnober allocation model assumes the computing system as a single super-server, which however may not always be the case. A real-world computing system (e.g., Google production cluster, Amazon EC2) generally consists of many heterogeneous servers with different resource capacities connected via a high-speed network. In this case, scheduling tasks efficiently to the computing system is analogous to the *NP-hard* multi-dimensional knapsack problem [10] mentioned above.

Affinity-based Task Scheduling Approach. We develop a heuristic approach for efficiency-oriented allocation by defining *affinity* of a task relative to the system. That is, when there are some idle resources on a server, we first filter out a set of pending tasks that can be accommodated by that server. Specifically, we first do the fairness-oriented allocation by looking for the user with the lowest dominant resource. If the soft fairness of that user is not guaranteed, then we consider all pending tasks from that user. Otherwise, we consider all pending tasks from all active users. Next, we compute the affinity score for each of these tasks, as the dot product between the task's resource demand and the vector of that server's idle resources. The one with the highest affinity score among all these pending tasks is chosen for scheduling.

6.2 Implementation

Figure 5 overviews the architecture design of QKYARN in YARN. We add four new components on top of YARN Resource Manager, namely, *QueueInfo Tracker (QT)*, *Maximum Dominant Resource Share (MDRS) Updater*, *QKnober Policy*, and *QKnober Resource Allocator*. QT tracks and monitors the task resource allocation information for each queue. MDRS Updater periodically updates the maximum dominant resource share for each user. With the provided runtime allocation information and maximum dominant resource share, QKnober Policy decides which resource allocator should be chosen for resource allocation, after which Resource Allocator allocates multiple resources to queues dynamically.

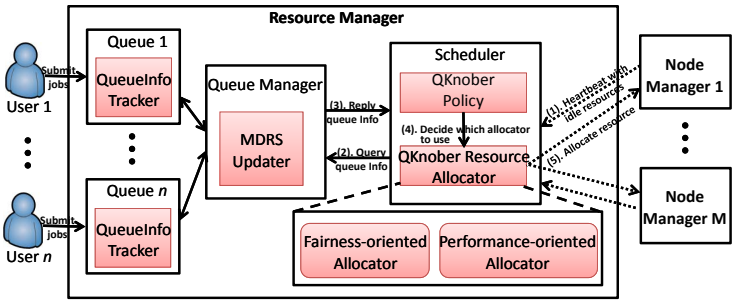


Fig. 5: QKYARN Architecture Overview. New components are added and shown in rectangle with pink background color, and others are from YARN.

Algorithm 1 Pseudocode for QKnober Allocation.

1: $\mathbf{R} = \langle r_1, ..., r_m \rangle$**:** total resource capacities.
2: $\mathbf{U}_i = \langle u_{i,1}, ..., u_{i,m} \rangle (i \in [1, n])$**:** currently allocated resources for user i.
3: s_i: current dominant resource share for user i.
4: s_i^{max}: the estimated maximum resource share for user i under DRF policy.
5: $W = \langle w_1, ..., w_n \rangle$: weighted share. w_i denotes the weight for user i.
6: $\rho (0 \leqslant \rho \leqslant 1)$: the knob argument, provided by users.

7: **while** there are pending tasks **do**
8: **Find** active user i with lowest dominant resource share s_i.
9: **if** $s_i < s_i^{max} \cdot \rho$ **then** ▷ Fairness-oriented allocation.
10: Allocate resource $\mathbf{D}_{i,k}$ to that task of user i.
11: **else** ▷ Efficiency-oriented allocation.
12: **Find** user j whose task k of demand $\mathbf{D}_{i,k}$ best fits for efficiency
13: optimization allocation among all users' tasks.
14: Allocate resource $\mathbf{D}_{j,k}$ to that task of user j.

QueueInfo Tracker (QT): As a multi-tenant system, YARN organizes resources into multiple queues, each of which represents a user or an organization. Users submit jobs to the queues. To enable the fairness-efficiency allocation across queues, QKYARN needs to maintain the runtime allocation information, which is achieved by inserting a component called Queue Tracker (QT) into each queue. It tracks the resource allocations of running and completed tasks for each queue. Moreover, it maintains the average task resource demand for each queue, based on the sliding window proposed in Section 6.1.

MDRS Updater: It is responsible for periodically updating the maximum dominant resource share s_i^{max} needed by QKnober Policy. Given a time interval Δt, it takes the average task resource demand of each queue provided by QT as input. The update is then triggered based on Formula (10) whenever $t_{curr} \% \Delta t = 0$, where t_{curr} represents the current time.

QKnober Policy: It is a key component of QKYARN. Being as a fairness-efficiency tradeoff scheduler, QKYARN relies on it to decide whether to perform fairness-oriented allocation or efficiency-oriented allocation at runtime. Algorithm 1 shows the pseudocode for QKnober allocation. Whenever there are pending tasks and idle resources available, QKnober will pick up the active user with the lowest dominant resource share (Line 8), of which the time complexity is $O(n)$. It first checks whether its dominant resource exceeds the dominant resource share of soft fairness(Line 9). If not exceeded, it will perform the fairness-oriented allocation with DRF (i.e., $O(n)$). Otherwise, it does the efficiency-oriented allocation by resolving the linear integer programming optimization problem of Formula (15) with GLPK [5] (Line 11). It adopts the simplex algorithm to solve the linear integer programming, of which the worst-case time complexity can be exponential. Fortunately, the number of constraints k (i.e., the number of different resource types) in Formula (17) is small (e.g., $k = 2$ in this paper) and the number of active users n is generally not large in practice, making the computing matrix for simplex algorithm is small and thus efficient in computation (See overhead evaluation in Appendix F of the supplemental material).

7 Experimental Evaluation

We use two complementary methods to evaluate the effectiveness of our proposed approach. We first evaluate QKnober using our prototype QKYARN on an Amazon EC2 cluster. To evaluate QKnober at larger scale, we perform trace-driven simulations using Google cluster-usage traces.

7.1 Experimental Setup

Hadoop Cluster. We have implemented QKnober in the version of YARN-2.4.0. We deploy the YARN framework in an Amazon EC2 cluster consisting of 60 Amazon EC2 t2.medium instances each with 2 virtual cores and 4 GB memory. We configure 1 instance as master, and the remaining 59 instances as slaves, each of which is configured with <2 virtual cores, 4 GB>.

Bin	Job Type	Map Tasks		Reduce Tasks		# Jobs
		#	Demand	#	Demand	
1	rankings selection	1	<1,1 GB>	NA	NA	38
2	grep search	2	<1, 1.5 GB>	NA	NA	18
3	uservisits aggregation	10	<2, 0.5 GB>	2	<4,2 GB>	14
4	rankings selection	50	<4, 1 GB>	NA	NA	10
5	uservisits aggregation	100	<2, 1.5 GB>	10	<2, 2 GB>	6
6	rankings selection	200	<3, 2 GB>	NA	NA	6
7	grep search	400	<2, 1 GB>	NA	NA	4
8	rankings-uservisits join	400	<1, 2 GB>	30	<2, 0.5 GB>	2
9	grep search	800	<2, 0.5 GB>	60	<1, 3 GB>	2

TABLE 1: Job types and sizes for synthetic Facebook workloads.

Testbed Workloads. We run four data-intensive workloads for QKYARN:

- *Synthetic Facebook Workload:* We synthesize Facebook workload based on the distribution of jobs sizes and inter-arrival time at Facebook provided by Zaharia et. al. [38]. The workload consists of 100 jobs. We categorize them into 9 bins of job types and sizes, as listed in Table 1. It is a mix of large number of small-sized jobs ($1 \sim 15$ tasks) and small

number of large-sized jobs (e.g., 800 tasks[1]). The job submission time is derived from one of SWIM's Facebook workload traces (e.g., FB-2009_samples_24_times_1hr_1.tsv) [3]. The demand distribution of map/reduce tasks is based on Figure 1 provided by Ghodsi et al [18]. The jobs are from Hive benchmark [1], containing four types of applications, i.e., rankings selection, grep search (selection), uservisits aggregation and rankings-uservisits join.

• *Purdue Workload:* Five benchmarks (e.g., WordCount, TeraSort, Grep, InvertedIndex, HistogramMovices) are randomly chosen from Purdue MapReduce Benchmarks Suite [7]. We use 40G wikipedia data [6] for WordCount, InvertedIndex and Grep, 40G generated data for TeraSort and HistogramMovices with provided tools. To emulate a series of regular job submissions in a data warehouse, we submit these jobs sequentially at an interval of 3 mins to the system.

• *Spark Workload:* We choose two machine learning algorithms, namely, kmeans and alternating least squares (ALS) with provided example benchmarks. We ran 10 instances of each algorithm, which are launched by a script that waits 2 minutes after each job completed to submit the next. We configure each kmeans instance with 100 workers, each with <2 CPUs, 2 GB> resources. In contrast, each ALS instance is set with 100 works, each with <1 CPU, 2 GB>.

• *TPC-H Workload:* To emulate continuous analytic query, such as analysis of users' behavior logs, we ran TPC-H benchmark queries on Hive [2]. 40 GB data are generated with provided data tools. Four representative queries Q1, Q9, Q12, and Q17 are chosen, each of which we create five instances. We launch one query after the previous one finished in a round robin fashion.

Trace-driven Simulator. To evaluate H-MRF at a larger scale, we developed a trace-driven simulator that replays logs from Google clusters. The simulator mimics these aspects of tasks from the original trace: task submission time, task resource requirements (e.g., cpu, memory and disk) and execution time, which are the least required information needed for any task scheduling simulator.

Trace Dataset. Originally, the Google traces provide the information about tasks submitted by over 900 users on a cluster of $12K$ machines in one month, which are specified by *job_events*, *task_events*, *machine_events*, *machine_attributes*, *task_constraints*, *task_usage* listed in the schema.csv file. The data of our simulator are retrieved from *task_events (user, task_index, cpu_request, memory_request, disk_space_request, · · ·)* and *task_usage (user, task_index, start_time, end_tme)* tables, both of which contain a common attribute of *user* and *task_index*. To generate our dataset, we first sort *task_events* according to *user* attribute. Next, we select tasks of the first one hundred users from *tasks_events* and find the corresponding task *start_time* and *end_time* from *task_usage* according to *task_index*. The *execution time* is calculated as *end_time* minus *start_time*. However, the Google trace does not provide the *task submission time*. In order to make simulator work, we make an assumption by setting *start_time* as *task submission time*.

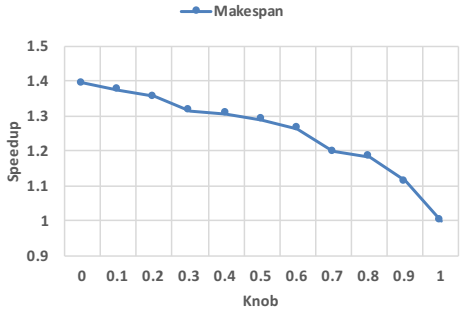
1. We reduce the size of the largest jobs in [38] to have the workload fit our cluster size.

7.2 Testbed Experimental Results

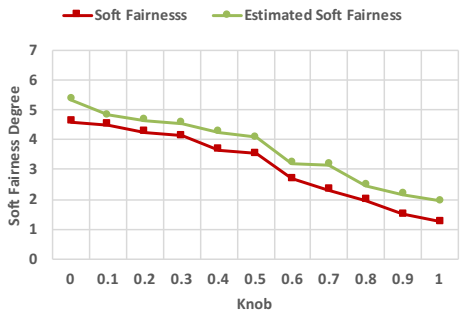
This section first evaluates the fairness and efficiency of QKnober under different knob values. Next we compare the performance of QKnober with its alternatives DRF and Tetris. Finally, we show the overhead of our QKnober system in Appendix F of the supplemental material.

7.2.1 Fairness vs. Efficiency

We show in Section 5 that QKnober is an elastic knob-based tradeoff allocation policy that allows users to balance the fairness and efficiency flexibly. In this section, we evaluate the impact of different knob values on the fairness and efficiency with the mix of four workloads experimentally. Suppose that there are four users A, B, C, D with the weighted shares of $1:2:3:4$, each running Facebook, Purdue, Spark and TPC-H workloads on the YARN cluster, respectively. With QKnober policy, we can then maximize the system efficiency while guaranteeing the soft fairness. We define a term called *soft fairness degree* to quantify the soft fairness based on Formula (12). The smaller soft fairness degree indicates the better fairness, and vice versa.



(a) Speedup.



(b) Soft Fairness.

Fig. 6: The system efficiency and soft fairness for QKnober under different knobs, where the speedup of Makespan is normalized over the case of knob $\rho = 1$, and the *estimated soft fairness* is computed according to Theorem 1. Figure 6(b) shows that the estimated soft fairness is close to the (experimental) soft fairness, implying that in practice we can instead tune the estimated soft fairness directly for the corresponding knob and use that knob value to perform the fairness-efficiency optimizstion with QKnober.

Figure 6 presents the experimental results (i.e., Makespan and Soft Fairness) as well as estimated soft fairness according to Theorem 1 for QKnober policy under different knob configurations. We compute the speedup based on the case when the knob is one. The larger value indicates the better performance. It can be observed that there is a strong tradeoff between fairness and efficiency. When the knob is small, it benefits the system efficiency but harms the fairness. In contrast, when we increase

the knob value, the fairness can become better at the cost of system efficiency. It means that users can make their own tradeoff preference over the fairness and efficiency by tuning the knob value.

Moreover, the computed soft fairness is much close to the (experimental) soft fairness, indicating that we can estimate the soft fairness in practice with Theorem 1. It can be important for users who want to tune the soft fairness θ directly instead of the knob ρ in practice. That is, given a certain desirable value θ of soft fairness, it can figure out the corresponding value ρ (i.e., estimate x-axis value according to y-axis value in Figure 6(b)). Under that knob value of ρ, it can then maximize the efficiency allocation with QKnober. (e.g., Figure 6(a)).

7.2.2 Performance Evaluation

Figure 7 (a) gives the normalized performance results for Static Partitioning, DRF, Tetris and QKnober under different knob configurations, where the speedup is computed over that of Static Partitioning on makespan. Particularly, we implement the static partitioning policy by dividing the whole cluster resources (e.g., CPU and memory) into four isolated portions for four workloads according to their weights mentioned in Section 7.2.1, and let them run exclusively without sharing. We have the following observations:

First, resource sharing (e.g., DRF, Tetris and QKnober) performs better than non-sharing (e.g., Static Partitioning). For fairness-only policy DRF, there is about 10% performance improvement over Static Partitioning. In contrast, for fairness-efficiency policies like Tetris and QKnober, the improvement can be up to 57% as we decrease the knob value from one to zero. The performance gain is mainly due to the resource preemption of unused resources from overloaded users in the sharing case, making the resource utilization higher than the non-sharing case. As illustrated in Figure 7 (b), the resource utilizations for sharing policies (e.g., DRF, Tetris, QKnober) are higher than that of static partitioning. For example, the average cpu utilizations for DRF, Tetris and QKnober are $55\%, 57\%$ and 69%, respectively, higher than the static partitioning of 46%.

Second, QKnober outperforms other baseline allocation policies DRF and Tetris in all knob configurations. Particularly, the reason why QKnober is better than DRF even when the knob is one is due to its efficient affinity-based task placement in reducing the fragmentation of machines in multi-resource allocation, whereas DRF policy does not have such a concern and simply views all machines as a single super machine. Moreover, both Tetris and QKnober are knob-based fairness-efficiency allocation policies. The reason why QKnober performs better than Tetris is due to their different approaches in the efficiency-oriented allocation. Tetris takes heuristic bin packing approach, whereas QKnober adopts the optimal linear integer programming method. It makes QKnober achieve a higher resource utilization than Tetris as shown in Figure 7 (b). Moreover, the performance improve as we decrease the knob from one to zero as illustrated in Figure 7 (a). This is because it benefits the efficient-oriented allocation as we decrease the knob, making it have a better resource utilization as shown in Figure 7 (c).

7.3 Google Trace Driven Simulation Results

In this section, we evaluate QKnober at a larger scale using Google cluster-usage traces [4]. We start by evaluating fairness and efficiency under different knobs using our built simulator to replay the execution of tasks from a Google cluster. Next we compare the performance among different scheduling policies under different numbers of users.

7.3.1 Evaluation on Different Knobs

In Section 7.2.1, we have shown the fairness-efficiency results for QKnober in a cluster shared by only four users. In this section, we evaluate fairness and efficiency for QKnober at a larger scale using Google cluster-usage traces. The distribution for tasks with respect to different resource demands is illustrated in Figure **??**. We simulate 100 users submitting tasks with different resource demands for three resource types (i.e., CPU, memory, and disk) in 24 hours to a Google cluster of 1000 machines, consisting of 527.5 CPU units, 513.2 memory units and 520.0 disk units in total, based on the Google trace's provided capacity information about machines. We assume that the resources of the cluster are equally shared across users.

Figure 8 illustrates the simulation results for QKnober under different knobs, where the speedup is computed over the case of $\rho = 1$. It shows that, it favors the speedup (efficiency) but worsens the fairness (i.e., the soft fairness degree is large) when knob value is very small. For example, there can be 26.7% performance improvement over the case of $\rho = 1$, but at the same time the soft fairness degree is as high as 4.26. In contrast, as we increase the knob value, the fairness becomes better at the expense of efficiency. In Figure 8, the soft fairness degree reduces to 2.71 when $\rho = 1$. In summary, all of these tradeoff observation results are consistent with those in Section 7.2.1.

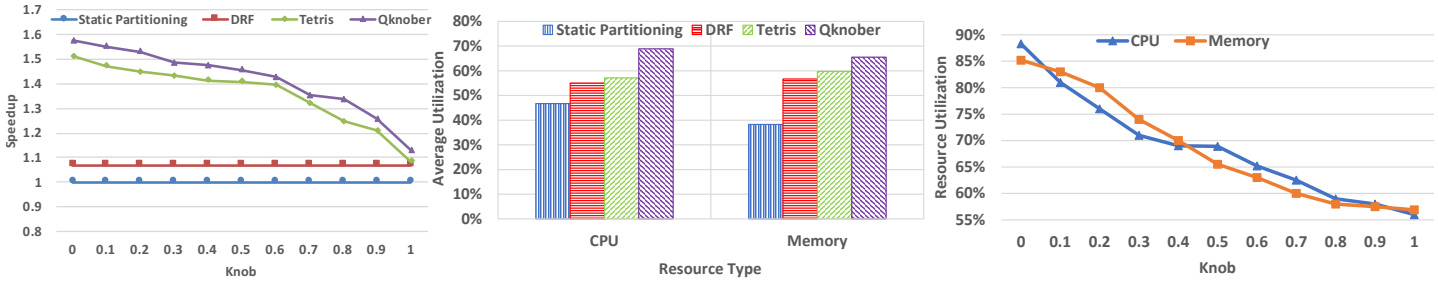
7.3.2 Evaluation on Different Numbers of Users

This section evaluates the efficiency for various scheduling policies under different numbers of users. Figure 9 presents the speedup results for four scheduling policies under different numbers of users, where the speedup is computed over that of Static Partitioning and the knob value of Tetris and QKnober are 0.2. There are several observations as follows.

First, for each fair scheduling policy (e.g., DRF, Tetris and QKnober), there is a decreasing trend of speedup results as we increase the number of users. For example, the speedup for QKnober decreases from $1.21\times$ to $1.10\times$ as we increase the number of users from 100 to 600. The behind reason is that, the resource contention becomes more serious when there are more users, which in turn leads to a lower resource utilization.

Second, sharing policies (e.g., DRF, Tetris and QKnober) is more efficient than non-sharing (e.g., Static partitioning) policy for all numbers of users. It can be observed in Figure 9 that all speedup results are larger than one for sharing policies. This is because resource sharing can allow idle unused resources from underloaded users to be utilized by overloaded users, which can improve the system utilization over the non-sharing case.

Third, Tetris works better than DRF, whereas QKnober outperforms Tetris for all numbers of users. Note that DRF is a pure fairness policy, which does not take into account the efficiency



(a) Speedup of makespan (over Static Partitioning). (b) Average resource utilization when the knob $\rho = 0.5$. (c) Average resource utilization for QKnober under different knobs.

Fig. 7: The comparison results of performance and resource utilization for Static Partitioning (i.e., non-sharing case), DRF, Tetris and QKnober under different knob configurations, where the speedup of makespan is computed over that of Static Partitioning.

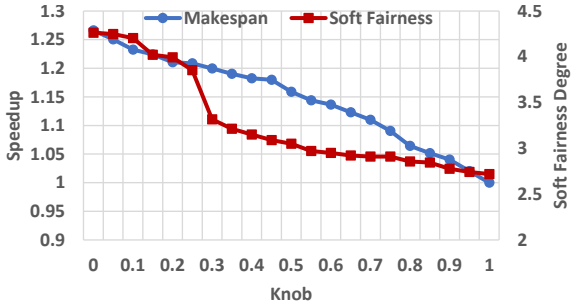


Fig. 8: The fairness-efficiency simulation results for QKnober under different knobs. We compute the speedup over the case of knob $\rho = 1$.

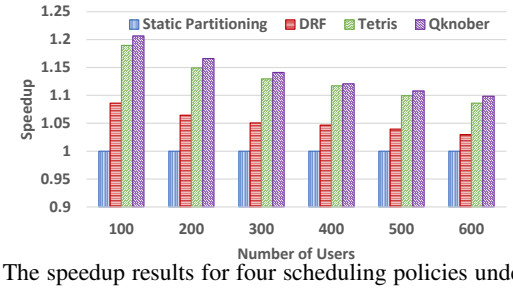


Fig. 9: The speedup results for four scheduling policies under different numbers of users, where the speedup is computed relative to that of Static Partitioning and the knob is $\rho = 0.2$.

optimization in its resource allocation. In contrast, both Tetris and QKnober are knob-based fairness-efficiency policies, which can trade some fairness for improved performance when $0 \leqslant \rho < 1$. In this case, the knob is $\rho = 0.2$, which is close to zero. As we have illustrated in Section 7.2.1 and 7.3.1, the system favors the efficiency when the knob is smaller. Moreover, although both Tetirs and QKnober adopt the knob-based approach, they take different optimization methods in their efficiency-oriented allocation. Tetris uses the bin-packing heuristics whereas QKnober utilizes the *optimal* liner programming method, making Qknober achieve better performance than Tetris.

8 Conclusion

This work studies the tradeoff between fairness and efficiency for users in a shared computing system. A preliminary version of this paper appears as [29]. We show that the knob-based approach is a promising solution to achieving the *flexible* and *elastic* tradeoff balance for users. However, existing knob-based fairness-efficiency schedulers are not aware of fairness degradation during its fairness-efficiency allocation, which either fail to guarantee δ-fairness or violate desired properties in Section 3. To address it, we develop a new knob-based fairness-efficiency policy called QKnober. It is a fairness sensitive scheduler that allows users to balance the fairness and efficiency with a knob while guaranteeing δ-soft fairness. Typically, we provably show that it meets several desirable properties including sharing incentive, envy freeness and pareto efficiency. Finally, we implement QKnober in YARN and our experiments show the promised initial results for QKnober that 1) there can be up to 57% performance improvement as we decrease the knob value for QKnober; 2) QKnober is better than DRF and Tetris by about 31.2% and 4.5%.

So far, this paper focuses only on the batch jobs in a datacenter by considering only CPU and memory resources. In future, we plan to study the fairness-efficiency job scheduling for multiple streaming jobs whose computing data are arriving over time. Moreover, we also consider the job scheduling for deep learning model training on a GPU-based environment.

Acknowledgement

This work is sponsored by the National Natural Science Foundation of China (61972277) and Tianjin Natural Science Foundation (18JCZDJC30800). Ce Yu is supported by the Joint Research Fund in Astronomy (U1731243, U1931130) under cooperative agreement between the National Natural Science Foundation of China (NSFC) and Chinese Academy of Sciences (CAS).

References

[1] Apache hive performance benchmarks. In *https://issues.apache.org/jira/browse/HIVE-396*, 2009.
[2] Apache tpc-h benchmark on hive. In *https://issues.apache.org/jira/browse/HIVE-600*, 2009.
[3] Facebook workload traces. In *https://github.com/SWIMProjectUCB/SWIM/wiki/Workloads-repository.*, 2009.
[4] Google cluster data. In *https://code.google.com/p/googleclusterdata/*, 2011.
[5] Glpk (gnu linear programming kit). In *https://www.gnu.org/software/glpk/*, 2012.
[6] Puma datasets. In *http://web.ics.purdue.edu/~fahmad/datasets.htm.*, 2012.
[7] Faraz Ahmad, Seyong Lee, Mithuna Thottethodi, and T. N. Vijaykumar. Puma: Purdue mapreduce benchmarks suite. In *ECE Technical Reports*, 2012.
[8] Arka A. Bhattacharya, David Culler, Eric Friedman, Ali Ghodsi, Scott Shenker, and Ion Stoica. Hierarchical scheduling for diverse datacenter workloads. In *Proceedings of the 4th Annual Symposium on Cloud Computing*, SOCC '13, pages 4:1–4:15, New York, NY, USA, 2013. ACM.
[9] R. P. Brent. Efficient implementation of the first-fit strategy for dynamic storage allocation. *ACM Trans. Program. Lang. Syst.*, 11(3):388–403, July 1989.
[10] Paul C Chu and John E Beasley. A genetic algorithm for the multidimensional knapsack problem. *Journal of heuristics*, 4(1):63–86, 1998.

[11] E. Danna, S. Mandal, and A. Singh. A practical algorithm for balancing the max-min fairness and throughput objectives in traffic engineering. In *INFOCOM'12*, pages 846–854, March 2012.
[12] Jeffrey Dean and Sanjay Ghemawat. Mapreduce: Simplified data processing on large clusters. In *OSDI'04*, pages 10–10, Berkeley, CA, USA, 2004. USENIX Association.
[13] Jeffrey Dean and Sanjay Ghemawat. Mapreduce: Simplified data processing on large clusters. *Commun. ACM*, 51(1):107–113, January 2008.
[14] Christina Delimitrou and Christos Kozyrakis. Quasar: Resource-efficient and qos-aware cluster management. In *ASPLOS '14*, pages 127–144, New York, NY, USA, 2014. ACM.
[15] Danny Dolev, Dror G. Feitelson, Joseph Y. Halpern, Raz Kupferman, and Nathan Linial. No justified complaints: On fair sharing of multiple resources. In *ITCS '12*, pages 68–75, NY, USA, 2012. ACM.
[16] Lars George. *HBase: the definitive guide*. " O'Reilly Media, Inc.", 2011.
[17] Ali Ghodsi, Vyas Sekar, Matei Zaharia, and Ion Stoica. Multi-resource fair queueing for packet processing. In *SIGCOMM '12*, pages 1–12, NY, USA, 2012. ACM.
[18] Ali Ghodsi, Matei Zaharia, Benjamin Hindman, Andy Konwinski, Scott Shenker, and Ion Stoica. Dominant resource fairness: Fair allocation of multiple resource types. In *NSDI'11*, pages 323–336, Berkeley, CA, USA, 2011. USENIX Association.
[19] Robert Grandl, Ganesh Ananthanarayanan, Srikanth Kandula, Sriram Rao, and Aditya Akella. Multi-resource packing for cluster schedulers. In *SIGCOMM'14*, pages 455–466. ACM, 2014.
[20] Benjamin Hindman, Andy Konwinski, Matei Zaharia, Ali Ghodsi, Anthony D. Joseph, Randy Katz, Scott Shenker, and Ion Stoica. Mesos: A platform for fine-grained resource sharing in the data center. In *NSDI'11*, pages 295–308, Berkeley, CA, USA, 2011. USENIX Association.
[21] Carlee Joe-Wong, Soumya Sen, Tian Lan, and Mung Chiang. Multiresource allocation: Fairness-efficiency tradeoffs in a unifying framework. *IEEE/ACM Trans. Netw.*, 21(6):1785–1798, December 2013.
[22] Ian Kash, Ariel D. Procaccia, and Nisarg Shah. No agent left behind: Dynamic fair division of multiple resources. In *AAMAS '13*, pages 351–358, 2013.
[23] Haikun Liu and Bingsheng He. Reciprocal resource fairness: Towards cooperative multiple-resource fair sharing in iaas clouds. In *SC '14*, pages 970–981, Piscataway, NJ, USA, 2014. IEEE Press.
[24] David C. Parkes, Ariel D. Procaccia, and Nisarg Shah. Beyond dominant resource fairness: Extensions, limitations, and indivisibilities. *ACM Trans. Econ. Comput.*, 3(1):3:1–3:22, March 2015.
[25] Charles Reiss, Alexey Tumanov, Gregory R. Ganger, Randy H. Katz, and Michael A. Kozuch. Heterogeneity and dynamicity of clouds at scale: Google trace analysis. In *SoCC '12*, pages 7:1–7:13. ACM, 2012.
[26] S. Tang, Z. Niu, B. He, B. Lee, and C. Yu. Long-term multi-resource fairness for pay-as-you use computing systems. *IEEE Transactions on Parallel and Distributed Systems*, 29(5):1147–1160, 2018.
[27] Shanjiang Tang, Qifei Chai, Ce Yu, Yusen Li, and Chao Sun. Balancing fairness and efficiency for cache sharing in semi-external memory system. 2020.
[28] Shanjiang Tang, BingSheng He, Shuhao Zhang, and Zhaojie Niu. Elastic multi-resource fairness: Balancing fairness and efficiency in coupled cpu-gpu architectures. In *SC '16*, pages 75:1–75:12. IEEE Press, 2016.
[29] Shanjiang Tang, Ce Yu, Chao Sun, Jian Xiao, and Yinglong Li. Qknober: A knob-based fairness-efficiency scheduler for cloud computing with qos guarantees. In Claus Pahl, Maja Vukovic, Jianwei Yin, and Qi Yu, editors, *Service-Oriented Computing*, pages 837–853, Cham, 2018.
[30] A. Thusoo, J.S. Sarma, N. Jain, Zheng Shao, P. Chakka, Ning Zhang, S. Antony, Hao Liu, and R. Murthy. Hive - a petabyte scale data warehouse using hadoop. In *ICDE'10*, pages 996–1005, March 2010.
[31] Hal R Varian. Equity, envy, and efficiency. *Journal of economic theory*, 9(1):63–91, 1974.
[32] Vinod Kumar Vavilapalli, Arun C. Murthy, Chris Douglas, Sharad Agarwal, Mahadev Konar, Robert Evans, Thomas Graves, and Lowe. Apache hadoop yarn: Yet another resource negotiator. In *SOCC '13*, pages 5:1–5:16, New York, NY, USA, 2013. ACM.
[33] Hui Wang and Peter Varman. Balancing fairness and efficiency in tiered storage systems with bottleneck-aware allocation. In *FAST'14*, pages 229–242, Berkeley, CA, USA, 2014. USENIX Association.
[34] Wei Wang, Chen Feng, Baochun Li, and Ben Liang. On the fairness-efficiency tradeoff for packet processing with multiple resources. In *CoNEXT '14*, pages 235–248, New York, NY, USA, 2014. ACM.
[35] Wei Wang, Baochun Li, and Ben Liang. Dominant resource fairness in cloud computing systems with heterogeneous servers. In *INFOCOM, 2014 Proceedings IEEE*, pages 583–591, April 2014.
[36] Wei Wang, Baochun Li, Ben Liang, and Jun Li. Multi-resource fair sharing for datacenter jobs with placement constraints. In *SC '16*, pages 86:1–86:12, Piscataway, NJ, USA, 2016. IEEE Press.
[37] Wei Wang, Shiyao Ma, Bo Li, and Baochun Li. Coflex: Navigating the fairness-efficiency tradeoff for coflow scheduling. In *INFOCOM'17*.
[38] Matei Zaharia, Dhruba Borthakur, Joydeep Sen Sarma, Khaled Elmeleegy, Scott Shenker, and Ion Stoica. Delay scheduling: A simple technique for achieving locality and fairness in cluster scheduling. In *EuroSys'10*, pages 265–278, New York, NY, USA, 2010. ACM.
[39] Matei Zaharia, Mosharaf Chowdhury, Michael J Franklin, Scott Shenker, and Ion Stoica. Spark: cluster computing with working sets. In *Hot Cloud'10*, volume 10, page 10, 2010.
[40] Seyed Majid Zahedi and Benjamin C. Lee. Ref: Resource elasticity fairness with sharing incentives for multiprocessors. In *ASPLOS '14*, pages 145–160. ACM, 2014.

Shanjiang Tang received the PhD degree from School of Computer Engineering, Nanyang Technological University, Singapore in 2015, and the MS and BS degrees from Tianjin University (TJU), China, in Jan 2011 and July 2008, respectively. He is currently an associate professor in College of Intelligence and Computing, Tianjin University, China. His research interests include parallel computing, cloud computing, big data analysis, and machine learning. He has published many papers in TCC, TPDS, TKDE, TSC, SC, ICS, HPDC, etc.

Yusen Li received the Ph.D. degree from Nanyang Technological University in 2014. He is currently an Associate Professor with the Department of Computer Science and Security, Nankai University, China. His research interests include scheduling, load balancing, and other resource management issues in distributed systems and cloud computing

Ce Yu received his P.h.D degree of Computer Science from Tianjin University(TJU) in 2009, MS, BS degree in 2005 and 2002 in the same University respectively. He is currently an associate professor and director of High Performance Computing Lab(HPCL) of Computer Science&Technology in Tianjin University. His main research interests include Parallel computing, Astronomy Computing, Cluster technology, Cell BE, MultiCore, Grid computing.