Scaling Large Production Clusters with Partitioned Synchronization

Yihui Feng, *Alibaba Group;* Zhi Liu, Yunjian Zhao, Tatiana Jin, and Yidi Wu, *The Chinese University of Hong Kong;* Yang Zhang, *Alibaba Group;* James Cheng, *The Chinese University of Hong Kong;* Chao Li and Tao Guan, *Alibaba Group*

https://www.usenix.org/conference/atc21/presentation/feng-yihui

This paper is included in the Proceedings of the
2021 USENIX Annual Technical Conference.

July 14–16, 2021

978-1-939133-23-6

Scaling Large Production Clusters with Partitioned Synchronization

Yihui Feng[*]
Alibaba Group

Zhi Liu[*†]
The Chinese University of Hong Kong

Yunjian Zhao[†]
The Chinese University of Hong Kong

Tatiana Jin[†]
The Chinese University of Hong Kong

Yidi Wu[†]
The Chinese University of Hong Kong

Yang Zhang
Alibaba Group

James Cheng[†]
The Chinese University of Hong Kong

Chao Li
Alibaba Group

Tao Guan
Alibaba Group

Abstract

The scale of computer clusters has grown significantly in recent years. Today, a cluster may have 100 thousand machines and execute billions of tasks, especially short tasks, each day. As a result, the scheduler, which manages resource utilization in a cluster, also needs to be upgraded to work at a much larger scale. However, upgrading the scheduler — a central system component — in a large production cluster is a daunting task as we need to ensure the cluster's stability and robustness, e.g., user transparency should be guaranteed, and other cluster components and the existing scheduling policies need to remain unchanged. We investigated existing scheduler designs and found that most cannot handle the scale of our production clusters or may endanger their robustness. We analyzed one most suitable design that follows a shared-state architecture, and its limitations led us to a fine-grained staleness-aware state sharing design, called partitioned synchronization (ParSync). ParSync features the simplicity required for maintaining the robustness of a production cluster, while achieving high scheduling efficiency and quality in scaling. ParSync has been deployed and is running stably in our production clusters.

1 Introduction

In Alibaba, we operate large clusters each containing tens of thousands of machines. Every day, billions of tasks are submitted to, scheduled and executed in a cluster. A *cluster scheduler*, or *scheduler* for short, manages both machines and tasks in a cluster. Based on the resource requirements (e.g., CPU, memory, network bandwidth) of tasks, a scheduler matches tasks to appropriate resources using various scheduling algorithms and makes complex trade-offs among multiple scheduling objectives such as scheduling efficiency, scheduling quality, resource utilization, fairness and priority of tasks. The ability to balance these objectives largely depends on both technical and business factors, and thus varies from company to company and cluster to cluster.

Due to the rapid growth in our businesses in recent years, we have faced serious challenges in scaling our scheduler, which is a centralized architecture similar to YARN [44], as there have been substantially more tasks and machines in our clusters. Today, the size of some of our clusters is close to 100k machines and the average task submission rate is about 40k tasks/sec (and considerably higher in some months). This scale simply exceeds a single scheduler's capacity and an upgrade to a distributed scheduler architecture is inevitable.

In this paper, we present *a design for a distributed scheduler architecture that can handle the scale of our cluster size and task submission rate, while at the same time achieving low-latency and high-quality scheduling.* The design presented also satisfies two hard constraints (§2): *backward compatibility* and *seamless user transparency*. The scheduler's robustness and stability derived from these constraints are of vital importance for our production environment.

Cluster schedulers have been extensively studied [9, 13, 14, 17, 25, 30, 32, 36, 37, 39, 44, 45] and we discuss the suitability of existing schedulers for our cluster environment and workloads in §3. Among them, a *shared-state scheduler architecture*, described in *Omega* [39], is able to handle our cluster size and satisfy the two hard constraints because it requires minimal intrusive architectural changes. In Omega, a master maintains the **cluster state**, which indicates the availability of resources in each machine in the cluster. There are multiple schedulers, each of which maintains a local copy of the cluster state by synchronizing with the master copy *periodically*[1]. Each scheduler schedules tasks *optimistically* based on the (possibly stale) local state and sends resource requests to the

[*] Co-first-authors ordered alphabetically.
[†] This work was done when the authors were visiting Alibaba.

[1] Omega [39] assumes that there is no synchronization overhead and thus each scheduler synchronizes the entire local state with the master whenever it communicates with the master to commit a task. But the synchronization overhead is not negligible in our cluster as it has a large state (due to the large size of the cluster), which does not allow us to synchronize the state at every task commit because of the high task submission rate (much higher than those in [39]). Thus, we synchronize the state periodically.

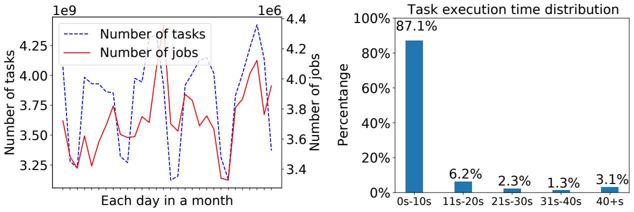
master. As multiple schedulers may request the same piece of resource, this results in **scheduling conflicts**. The master grants the resource to the first scheduler that requested it, and the other schedulers will need to reschedule their task. The scheduling conflicts and rescheduling overheads lead to high latency when the task submission rate is high, which we validate in §4 using both analytical models and simulations. Our results show that the *contention on high-quality resources* and the *staleness of local states* are the two main contributors to high scheduling latency as they both substantially increase scheduling conflicts.

Then in §5, we propose **partitioned synchronization** (**ParSync**) to mitigate the negative impacts of these two factors. ParSync partitions the cluster state into N parts (N is the number of schedulers), such that each scheduler synchronizes $1/N$ of distinct partitions instead of the entire state each time. As a result, at any time each scheduler has a fresh view of $1/N$ of the partitions and can first schedule its tasks to these partitions. This significantly reduces the contention (with other schedulers) on high-quality resources. Based on when a partition is synchronized, a scheduler knows how stale its partitions are; thus, algorithm designers can now better balance scheduling latency and scheduling quality by adjusting the preference to fresher partitions. We also devise an adaptive strategy to dynamically choose between lowering scheduling latency and improving scheduling quality. We validate the effectiveness of ParSync and various scheduling algorithms developed based on ParSync in §6. ParSync has been deployed in Fuxi 2.0, which is the latest version of the distributed cluster scheduler in Alibaba, and is running stably in our production clusters, where each production cluster contains thousands to 100K machines.

2 Background and Challenges

Workload statistics. Millions of jobs are submitted to a cluster each day in Alibaba. Figure 1a (solid curve) plots the number of jobs processed in a cluster each day in a randomly picked month, which ranges from 3.34 to 4.36 million jobs. A job consists of many tasks and Figure 1a (dotted curve) shows that the number of tasks each day ranges from 3.1 to 4.4 billion. The majority of tasks are short tasks. As shown in Figure 1b, 87% of tasks are completed in less than 10 seconds.

Motivation for scheduler upgrade. Our previous cluster scheduler follows a typical master-worker architecture [44]. A single *master* manages all the resources in a cluster and handles all the scheduling work. Each worker machine has an *agent* process, which sends the latest status of the worker via heart-beat messages to the master. The master receives jobs submitted by users and then places each job in its corresponding *quota group* [26][2]. The cluster operator configures a quota group to specify the minimum and maximum amounts of resources the group can acquire. In particular, when the cluster is overloaded, resources have to be divided among all the groups by weighted fairness (based on their quotas).



(a) Number of daily jobs/tasks (b) Task execution time

Figure 1: Job/task statistics

In recent years, the scale of our cluster has significantly increased and a single cluster can have 100k machines. *Statically partitioning a large cluster is not an option* because there are some extremely large jobs from critical projects and a large cluster is required to ensure that these large jobs and a large number of daily production jobs can be both processed without extended delay. There are also important technical reasons (e.g., resource fragmentation [45], limited visibility [39]) and business factors (e.g., projects need to access the data of other projects in the same business unit that stores its data in a single cluster for operational and management reasons), *which require scheduling over an entire large cluster rather than breaking it down into smaller ones*.

However, the previous monolithic single-master architecture could not handle the scale of the current clusters in Alibaba. First, the 10x larger number of machines requires a larger time gap between two consecutive heart-beat messages from a worker to the master so that the master, which is on the critical path of all decisions, would not be overloaded. But a larger gap is more likely to leave idle machines unused for a longer period of time during the gap. Second, the significantly larger number of tasks simply exceeds the capability of a single scheduler. Specifically, our previous scheduler could only handle task submission rates in the range of a few thousand tasks per second, but the average task submission rate is 40K/sec (and the peak is 61K/sec) for the 30 days in Figure 1a.

Objectives and constraints. In addition to the scalability challenges posed by both the workload and the cluster size, the new scheduler should also achieve a good trade-off among various scheduling objectives. We focus on the following objectives: (1) *scheduling efficiency or delay*, i.e., how long a task waits for its required resources to be allocated; (2) *scheduling quality*, i.e., whether the resource preference of a task (e.g., machines where its data are stored, machines with larger memory or faster CPUs) is satisfied; (3) *fairness* [19] and

[2] Jobs belong to projects and projects are assigned resource quotas according to their budgets. A *quota group* can be considered as a virtual cluster that is allocated with a certain amount of resources according to a project's available resource quotas.

priority; and (4) *resource utilization*. These objectives often contradict each other. For example, high scheduling quality may prolong the scheduling delay; maintaining strict fairness can leave resources unused and thus hurt utilization.

Over the years, the complicated logic for balancing these objectives has been programmed into various scheduling strategies. In addition, other cluster components such as application masters and worker agents are maintained by other teams in Alibaba, and it takes years' collaborative efforts to make them robust and behave as expected. Thus, *the new scheduler design should make as few changes to the existing codebase as possible, so as to ensure the entire system's robustness and backward compatibility (e.g., other cluster components need not be changed and the existing scheduling strategies can be applied in the same way). Finally, the system upgrade should be transparent to both the internal users and cloud clients of Alibaba.*

Methodology. We first investigate existing works that may serve as a potential solution and pick the most suitable one for further analysis. We then develop a simplified model to gain insights mathematically. As it is hard to acquire an accurate mathematical solution and the simplified model also leaves out critical factors that cannot be dismissed in a real production environment, we use a simulation to find out the determining factors, which are used to derive our solution to the scalability problem. Finally, we evaluate our solution in a high-fidelity simulation cluster with workloads sampled from the production clusters as shown in Figure 1b.

3 Can We Adopt an Existing Scheduler?

As many schedulers have been proposed, we first examine if our problem can be addressed by an existing scheduler.

3.1 Shared-State Architecture

Omega [39] proposed the *shared-state scheduler architecture* as described in §1. This architecture addresses the limitations in *monolithic*, *two-level*, and *static partitioning* approaches, respectively. First, each scheduler in the shared-state architecture can run a different scheduling strategy programmed in separate code bases for different types of jobs, as to avoid the software engineering difficulties of maintaining all strategies in one code base and using multi-threading for solving head-of-line blocking in *monolithic* architecture. Second, each scheduler has a global view of the cluster, thus solving the problem of limited visibility in *two-level* schedulers such as Mesos [25]. This allows global policies (e.g., fairness and priority) to be implemented. Third, each scheduler can assign tasks to any machine in the cluster instead of a fixed subset of partitions, which reduces resource fragmentation in *statically partitioned* clusters and achieves higher utilization [45].

The shared-state architecture is a neat design that also satisfies our two hard constraints given in §1. First, interactions between application masters, worker agents and schedulers would require little change. An application master still talks to only one scheduler and performs application-level scheduling on a pool of committed resources. An agent reports the status of each task to its corresponding scheduler in the same way. Thus, backward compatibility is ensured. Second, users rely on the behaviors of global policies such as fairness and priority to organize their workflows to ensure job completion before deadline and effective resource utilization. They may have accumulated many scripts for arranging job submissions over the years. With global policies unchanged, these scripts can be reused.

With the hard constraints resolved, we move on to analyze whether this architecture also meets our other design goals given in §1, i.e., scalability, low latency and high scheduling quality. We present our analysis in §4, but before that, we also consider other alternatives below.

3.2 Schedulers Focusing on Scalability

Shared-state approaches. Apollo [9], Mercury [30] and Yaq-d [37] also use a shared state. They do not resolve conflicts by a centralized coordinator as in Omega, but let schedulers push tasks to distributed queues in workers. Based on task duration estimation, Apollo shares a *wait-time matrix* (*WTM*) as the state to keep the queue length in each worker. The WTM allows a scheduler to infer future resource availability instead of being limited to present resource availability. Yaq-d discusses queue sizing and reordering strategies based on wait time. In Mercury, tasks with guaranteed resources are handled by a centralized scheduler in order to enforce fairness and priority, while other tasks are handled by distributed schedulers.

These schedulers are not as general as Omega's design (e.g., their designs cannot be easily adopted in cluster) for the following reasons. First, task duration estimation can be inaccurate due to little prior information, changing input data size, data skewness [11, 33] and temporal interference [8, 16]. Task duration estimation is particularly difficult for modern clusters (e.g., clouds, our data centers) as they process diversified tasks from both internal and external users running on a broad range of systems. Second, predicted resource availability improves scheduling quality primarily due to disk locality, e.g., in Apollo's cluster [9]. A scheduler would assign a task to a busy machine as long as the benefit from disk locality is greater than the extra time the task waits to be executed. However, disk locality becomes less important [7] when faster switches, NICs [1, 6] and compressed file format [2–5, 42] are in use. Third, distinguishing tasks as in Mercury complicates the intra-job scheduling design of the applications, which compromises backward compatibility.

Other approaches. There are also distributed schedulers that do not share a cluster state. Sparrow [36] uses batch sampling and late binding to improve scheduling efficiency and task completion time. Tarcil [17] extends Sparrow by dynamically adjusting sampling sizes. Hawk [14] dispatches long batch jobs to a centralized scheduler, which adopts the least-waiting-time strategy, and short jobs to a set of distributed schedulers. Hawk also reserves a small portion of the cluster for short jobs and uses task stealing to avoid short tasks being blocked by long tasks. Eagle [13] takes one step further to proactively avoid assigning a short task to a machine with long tasks running or waiting. It also enables a worker machine to repetitively fetch remaining tasks from a job to mimic the least-remaining-time-first strategy.

The designs of these schedulers are also not general and adopting them has the following key concerns. First, they also rely on accurate task duration estimation as in shared-state approaches. Second, Sparrow does not have a global view of a cluster to implement global policies, and its strategy is customized for fine-grained tasks of Spark jobs [35] and the duration of such tasks is in the range of milliseconds. Hawk and Eagle focus on using a single scheduling algorithm to reduce the JCT of homogeneous workloads. In contrast, Omega's design offers a global cluster view and allows multiple schedulers to run different scheduling algorithms.

3.3 Other Related Work

Schedulers such as YARN [44], Mesos [25], Borg [41,45] and Kubernetes [32] consider various issues in production clusters such as generality, extensibility, robustness and resource utilization, in contrast to schedulers discussed in §3.2 which focus on high scalability and low scheduling latency. Apart from scheduler architectures, many scheduling algorithms have been proposed. [23,29,31] focus on intra-job task scheduling to optimize job completion time. Job scheduling algorithms that aim to achieve objectives such as fairness [10,19,27,47], high resource utilization [12,15,16,20,20,21,27,28,34,40,46], job completion time [22,43], and workload autoscaling [38] may also be adopted in our schedulers according to the application needs of our clusters. These works do not focus on scheduler architecture design and are orthogonal to our work.

4 An Analysis on Scheduling Conflicts

In §3 we singled out Omega's shared-state architecture [39] as a potential solution to our problem. As mentioned in §1, *multiple schedulers may contend for the same piece of resource*, which leads to **conflicts** and hence scheduling delay. Thus, we want to study the following questions: *(1) What are the factors that determine the conflict rate? how important is each of these factors? (2) How bad can the scheduling delay be? (3) How can we avoid or alleviate the scheduling delay? What price do we need to pay?*

4.1 Conflict Modeling

We first quantify conflicts by constructing an analytical model for scheduling using the shared-state architecture presented in §1. The scheduling of a task may be delayed when there are not enough available resources to be allocated for the submitted tasks, or when there are enough available resources but the scheduler cannot keep up with the task submission rate. As we focus on the scheduler design, we only investigate the second case, i.e., the scheduler is the potential bottleneck of allocating available resources to tasks in time.

One extreme case that puts schedulers on the test is when the task submission rate and the resource needs of the tasks just match with the total amount of resources in a cluster. Suppose that there are S slots of resources in the cluster and each task takes one slot for its execution. At each unit of time (UT), J tasks are submitted for scheduling and each of the tasks runs for a fixed duration of T UT. If we have a single ideal scheduler that assigns tasks without delay and incurs no conflict, then all the J tasks will be committed immediately as long as there are available slots. We say that a task is **committed** when its requested resources are allocated. In reality, however, we do not have such an ideal scheduler. Instead, we have multiple schedulers that can lead to conflicts. By queuing theory, the scheduling delay will grow to infinity. However, as we will discuss below, this extreme case helps us see how much overhead due to conflicts is added by introducing multiple schedulers and how much price we should pay to fix this problem. We are particularly interested in this extreme case as it gives the scheduler the most pressure.

Now suppose that a scheduler can only schedule and commit $K < J$ tasks within each UT *given that there is no conflict*. To keep up with the task submission rate, i.e., J tasks/UT, we need to use at least $N = J/K$ schedulers to share the scheduling load. We assume that tasks are uniformly distributed to the schedulers and each scheduler independently makes its own scheduling decision based on the latest synchronized local cluster state, where available slots are randomly chosen for assignment. We will discuss the implications of these assumptions in §4.2.

We name the above setting as the **Baseline**. As conflicts cannot be avoided in reality, a scheduler cannot commit all its K tasks within the current UT when conflicts occur, which leads to scheduling delay. We investigate how many conflicts can be incurred. Consider that the cluster has S_{idle} idle slots. We denote the number of schedulers picking slot i as a random variable X_i. Since each scheduler picks a slot with probability $\frac{K}{S_{idle}}$, X_i follows a binomial distribution. We denote the number of conflicts at slot i as a random variable $Y_i = max(X_i - 1, 0)$.

Then, we can deduce the expectation of Y_i as:

$$
\begin{aligned}
\mathbb{E}(Y_i) &= 0 \cdot \Pr\{X_i \leq 1\} + \sum_{j=1}^{N-1} j \cdot \Pr\{X_i = j+1\} \\
&= \sum_{j=1}^{N-1} (j+1) \cdot \Pr\{X_i = j+1\} - \sum_{j=1}^{N-1} \Pr\{X_i = j+1\} \\
&= \mathbb{E}(X_i) - \Pr\{X_i = 1\} - (1 - \Pr\{X_i = 1\} - \Pr\{X_i = 0\}) \\
&= \frac{NK}{S_{idle}} - 1 + \left(1 - \frac{K}{S_{idle}}\right)^N \qquad (1)
\end{aligned}
$$

Since we have S_{idle} idle slots in total, the expectation of the total number of conflicts is given by:

$$
\mathbb{E}(\sum_{i=1}^{S} Y_i) = S_{idle} * \mathbb{E}(Y_i) = NK - S_{idle} + S_{idle} * (1 - \frac{K}{S_{idle}})^N \qquad (2)
$$

To reduce conflicts, naturally we may consider to add more slots (so as to reduce the contention for resources) or more schedulers (so as to increase the capacity to handle conflicts). We analyze each of them as follows.

Adding extra slots. Merely adding extra slots to a cluster can never reduce the expectation of the number of conflicts to 0 according to Eq.(2). We may reduce conflicts by adding more extra slots, but Eq.(2) shows that the effect of increasing the number of extra slots (i.e., S_{idle}) is superlinearly diminishing.

Adding extra schedulers. In the Baseline setting, each scheduler is operating at its full capacity and does not have room to resolve conflicts. By adding more schedulers, each scheduler may handle less than K tasks in each UT and now have time to reschedule tasks due to conflicts. However, this may lead to more conflicts, as the following analysis shows. Suppose now we have $A * N$ (where $A > 1$) instead of N schedulers, and thus each scheduler has $\frac{K}{A}$ tasks to schedule. Substituting N with $A * N$ and K with $\frac{K}{A}$ in Eq.(2), we obtain:

$$
\begin{aligned}
f(A) = \mathbb{E}(\sum_{i=1}^{S} Y_i) &= AN\frac{K}{A} - S_{idle} + S_{idle} * (1 - \frac{K}{AS_{idle}})^{AN} \qquad (3) \\
&= C_1 + C_2 * (1 - \frac{1}{x})^{x * C_3}, \qquad (4)
\end{aligned}
$$

where $C_1 = NK - S_{idle}$, $C_2 = S_{idle}$, $C_3 = \frac{NK}{S_{idle}}$, and $x = \frac{AS_{idle}}{K}$.

In Eq.(4), since C_1, C_2 and C_3 are independent of A and $C_2, C_3 > 0$, $f(A)$ increases if $(1 - \frac{1}{x})^x$ increases. Since $(1 - \frac{1}{x})^x$ increases monotonically as A increases when $x \geq 1$, $f(A)$ also increases as A increases, i.e., the expected number of conflicts increases when more schedulers are added. Thus, we need to find out *whether the overhead of having more conflicts can be covered up by the benefit of having more time for rescheduling tasks such that these tasks will be committed within the same UT in which they are submitted.*

Adding extra slots and schedulers. Due to the diminishing returns by adding extra slots only or schedulers only, it is reasonable to believe that adding both of them can lead to a more cost-effective solution. However, as S_{extra} and A are intertwined in Eq.(4) and Eq.(4) only indicates the number of conflicts in a single round of scheduling, it is hard to derive an accurate mathematical solution for S_{extra} and A to achieve 0 scheduling delay in a series of rounds of scheduling. Nevertheless, we can make use of the equations we have derived for quantifying conflicts to construct a simulator for the scheduling process. By simulating different combinations of N, J, K, S, with the constraints $J * T = S$ and $N * K = J$, we can examine the minimal requirements for S_{extra} and A.

4.2 The Implications of Our Assumptions

Before we present the simulation, we remark that although the assumptions made in §4.1 lead to an easier analysis, some of them diverge from the reality in the following aspects:

[A1] It is assumed that schedulers pick slots for tasks in a uniformly random fashion. In reality, some machines may be preferred (e.g., because of more advanced hardware or data locality) and result in more conflicts than the expectation.

[A2] It is (implicitly) assumed that all the schedulers schedule tasks synchronously round after round, and the local cluster states can be synchronized with the master state as frequently as we want. However, in reality each scheduler acts asynchronously with each other. We also cannot piggyback the synchronization of the cluster state on the return trip of every commit request as in Omega [39], as explained in Footnote 1 in §1. Instead, a time gap G is set between two synchronizations of a local state with the master. In-between the gap, when a scheduler commits a task to some slot, the slot's status in the master state and the scheduler's local state is updated. Such updates make the local states of other schedulers *stale*, and scheduling decisions based on a stale view of the state lead to more conflicts than indicated by our equations in §4.1.

[A3] We assume conflicts are uniformly incurred on schedulers. But in reality commit requests from schedulers are handled by the master in an FIFO manner and thus schedulers whose requests are sent earlier will get fewer conflicts.

4.3 Simulation Analysis

We construct a simulator following most of the setting in §4.1. We also integrate the factors listed in §4.2 into the simulator as follows.

To address [A1], we assign a score to each slot based on a normal distribution and schedulers pick desired slots by weighted sampling according to their scores (instead of in a uniformly random fashion as in §4.1). We vary the variance of slot scores to control how much slots should be contended.

To address [A2], we set a **synchronization gap** G. At each time period G, schedulers synchronize their local state with the master state. Each scheduler makes its scheduling decisions independently based on its own local state, which may become stale until its next synchronization. Since G controls the staleness of the local states, we vary G to examine its impact on scheduling delay.

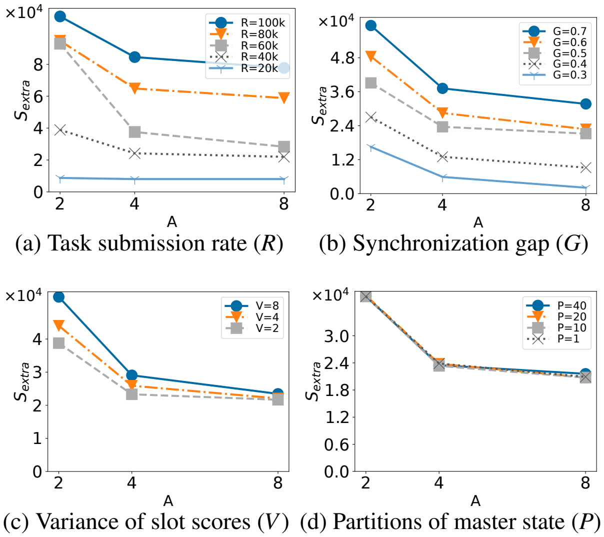
To address [A3], commit requests are sent to the master asynchronously by schedulers in our simulation and the requests are processed in FIFO. We also distribute the master state by partitioning so that commit requests to a slot are only sent to the partition that contains the slot. We vary the number of partitions to examine the effects of having a long queue at a single master versus shorter queues at distributed partitions.

Simulation setting. We assume that the cluster has 200K slots, and each task takes 1 slot and uses it for 5s, which is to simulate a typical scenario in our cluster, i.e., task submission rate is around 40K tasks/s and the duration of most tasks is between 1s to 10s (§2). Suppose that a scheduler takes 0.25ms to schedule a single task (close to our real scheduling latency) and thus we need at least 10 schedulers to just match with the 40K/s task submission rate (assuming no conflicts). Tasks are submitted in batches, where each batch has 100 tasks, and 10 batches are submitted to each scheduler in each second. Each scheduler makes scheduling decisions for a whole batch and schedules the next batch only after the tasks in the current batch have all been committed.

We vary task submission rate R, synchronization gap G, slot scores, and the number of partitions P of the master state, to examine how S_{extra} changes with $A = 2, 4, 8$. We want to find which factors are the main contributors to conflicts. For each simulation, we use the following *default setting*: $R =$ 40K/s, $G = 0.5$s, a normal distribution of slot scores with mean $= 10$ and variance $V = 2$, and $P = 1$ (i.e., no partitioning).

In the simulation, we try to find the minimum number of combinations of A and S_{extra} to achieve a small scheduling delay. With a larger A, the number of tasks B in one batch is adjusted to $B_{new} = \frac{B}{A}$. With a larger S_{extra}, the number of conflicts decreases. In the simulation, tasks are submitted at a fixed rate for 90 seconds (similar patterns are observed after 90s), and we say that the scheduling delay is small when the simulation finishes in less than 110% × 90s = 99s.

Simulation results. Figures 2a, 2b and 2c show that *a significant number of extra slots needs to be added as we increase R, G or V*. This is because *a larger R means more work for each scheduler, a larger G means making scheduling decisions based on a staler state and for longer time, and a larger V means more contentions for higher-quality slots, all of which lead to more conflicts. To maintain the same scheduling delay, more extra slots can help reduce the contentions for slots and hence conflicts*. However, Figure 2d shows that the impact of P is minimal. This is because even when the master state is not partitioned, some schedulers may have more conflicts at one point of time but the number of conflicts evens out among all schedulers over time.



(a) Task submission rate (R)

(b) Synchronization gap (G)

(c) Variance of slot scores (V)

(d) Partitions of master state (P)

Figure 2: Simulation results

In each simulation, we also increase the number of schedulers to 2, 4 and 8 times more to see how a larger A helps reduce scheduling delay. Figures 2a-2d consistently show that *increasing A has diminishing benefits. This conforms with our findings in §4.1 as sharing the same total amount of work by more schedulers leads to more conflicts.* Moreover, it is costly to operate many schedulers in a real cluster, as we use active standbys to replace failed ones to meet our SLAs.

Adding a large number of extra slots and schedulers is a big price to pay for a large-scale production cluster, considering both the extra machine costs and daily operation costs. Thus, we want to look for a new solution. According to the simulation results, task submission rate, slot scores and synchronization gap are the main factors that contribute to conflicts. However, we cannot change task submission rate as it is fixed by the workloads. Slot scores are related to the resources desired by tasks, which may be compromised (e.g., a task may also accept a fast machine instead of the fastest machine) to alleviate the contentions and hence reduce conflicts. Synchronization gap affects performance as it controls the staleness of the local states, which in turn determine the probability of conflicts. Our solution will focus on the last two factors, i.e., **slot quality** and **staleness of the local states**.

5 Partitioned Synchronization

As our solution aims to reduce conflicts, we first describe an alternative that offers no-conflict guarantee, namely *pessimistic scheduling*. One implementation of pessimistic scheduling is to let each scheduler have *exclusive partitions* of a cluster, but this can lead to resource under-utilization as some partitions may hold idle resources that can be used by other schedulers [39, 45]. Another implementation is by applying locks on resource slots, but this can block the actions of other schedulers and increase scheduling delay due to reduced concurrent processing [24, 39]. We want to avoid the limitations of pessimistic scheduling and reduce conflicts in optimistic scheduling (as in Omega), while enjoying their benefits, i.e., *high concurrent processing while having few conflicts*.

5.1 The Design

Observation. We found that scheduling delay increases disproportionately within the period G. When the cluster state is just synchronized, it is fresher and scheduling has fewer conflicts. But when the state becomes more outdated towards the end of G, scheduling decisions result in more conflicts. Conflicts lead to rescheduling, which may in turn cause new conflicts, and hence rescheduling recursively. Consider our default simulation setting in §4.3 and $S_{extra} = 15,000$, and divide G into two equal intervals: 0s-0.25s and 0.25s-0.5s. The average conflict rate in the first interval is 48% and that in the second interval is increased to 64%. As a result, the average delay of the first interval is only 23% of the total average delay, while the second interval contributes to 76% of the total. In other words, most of the delay is caused by the staler view of the cluster state in the later interval of G.

Main idea. Our main idea is *to reduce the staleness of the local states and to find a good balance between resource quality (i.e., slot score) and scheduling efficiency*, as these two are the main contributors to conflicts according to our findings in §4.3. We present our solution, **partitioned synchronization** (**ParSync**), as follows.

ParSync logically partitions the master state into P parts. Assume that we have N schedulers, where $N \leq P$. Each scheduler still keeps a local cluster state, but *different schedulers synchronize different partitions of the state each time*, in contrast to *synchronizing the entire state* (let us denote it as **StateSync**) as in the approach discussed in §4. Specifically, assume (for simplicity) P is a multiple of N, the i-th scheduler starts its synchronization from the $(\frac{P}{N} * (i-1) + 1)$-th partition to the $(\frac{P}{N} * i)$-th partition of the state in each round of synchronization, and the subsequent rounds of synchronization continue in a round-robin manner. Thus, in each round, all schedulers synchronize different partitions of the cluster state. This is also why we require $P \geq N$, as otherwise some schedulers would synchronize the same partition(s).

How does ParSync work? First, each scheduler has a *fresh* view of $\frac{P}{N}$ partitions of the cluster resources, so that the scheduler can commit its tasks to available slots in these partitions with a high success rate (since its view on these partitions is fresher than that of any other schedulers). This design is particularly favorable for scheduling short tasks in a highly contended cluster like ours. For short tasks, it is more critical to acquire resources sooner for their execution, instead of spending long time to find the most suitable slots because better slots may not compensate for the scheduling delay (which itself can be comparable with or even longer than the execution time of a short task). Note that the majority of the tasks in our workloads are short tasks, as shown in Figure 1b.

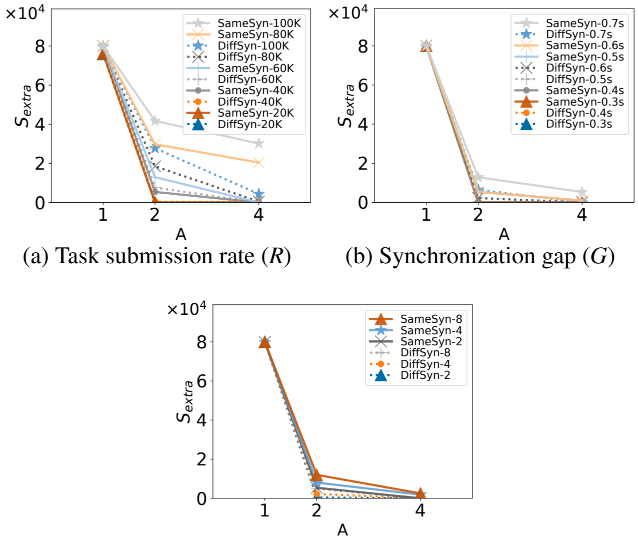
Second, ParSync also effectively reduces the average *staleness* of the entries in a local cluster state, which we explain as follows. As the granularity of synchronization is changed from the entire cluster state to $\frac{P}{N}$ partitions, which effectively changes the synchronization gap to $\frac{G}{N}$. In contrast to synchronizing the entire state at every G time units, with ParSync each scheduler synchronizes $\frac{P}{N}$ partitions of its local state at every $\frac{G}{N}$ time units. Now let us define the **staleness** of a partition as the period of time since its latest synchronization, and let **AS** be the **average staleness** of all the partitions of a local state. Whether ParSync is used or not, the average of "AS"s over all time points is always $\frac{G}{2}$. However, ParSync reduces the variance of "AS"s, which we explain as follows.

Suppose that there is only one partition, then the minimum and maximum AS are 0 and G (at time 0 and G, respectively). If there are two partitions, then the minimum and maximum AS are $(0 + \frac{G}{2})/2 = \frac{G}{4}$ and $(\frac{G}{2} + G)/2 = \frac{3G}{4}$. Thus, when we increase the number of partitions from 1 to 2, we also bound the AS in the range of $[\frac{G}{4}, \frac{3G}{4}]$ instead of $[0, G]$. Intuitively speaking, using more partitions here is like sacrificing the period when the AS is small (i.e., $[0, \frac{G}{4}]$) to avoid the period when the AS is large (i.e., $[\frac{3G}{4}, G]$). But *this sacrifice can lead to significant reduction in the scheduling delay, because conflicts increase much faster in the later interval of G* as we discussed in the Observation earlier.

As having more partitions bounds the AS in a smaller range, consequently the variance of AS is also reduced. When the number of partitions becomes sufficiently large, we push the minimum and maximum AS close to $\frac{G}{2}$. Thus, *the eventual effect of ParSync on scheduling delay can be approximately viewed as reducing G to $\frac{G}{2}$, which also means that we reduce the average staleness of a local state*.

The above analysis is important because our Observation earlier also indicates that a reduction in the staleness of the local state can significantly reduce the number of conflicts and hence also the scheduling delay.

ParSync allows us to better balance resource quality and scheduling efficiency. For example, as conflicts are less likely to happen in periods when the cluster is not in intensive use, a scheduler may take higher risk of rescheduling and try to commit tasks to high-quality slots in other staler partitions (instead of its freshest $\frac{P}{N}$ partitions). In this case, ParSync may adopt optimistic scheduling to prioritize resource quality (i.e., slot score). In contrast, a scheduling strategy aiming for successful commits may take a pessimistic approach (but with all schedulers concurrently working). In addition, each scheduler still has a global view of the cluster and hence global scheduling policies can also be implemented.



(a) Task submission rate (R)

(b) Synchronization gap (G)

(c) Variance of slot scores (V)

Figure 3: The effects of different synchronization strategies

5.2 Simulation Analysis on ParSync

We further analyze the effectiveness of ParSync by simulation. In all the simulations, we use the same setting and the default values of the parameters as in §4.3.

Synchronization strategies. We first compare two synchronization strategies: (1) **DiffSyn**: each scheduler synchronizes different partitions of the cluster state, i.e., the strategy used in ParSync; (2) **SameSyn**: all schedulers synchronize the same partitions, i.e., all schedulers synchronize the $(\frac{P}{N}*(i-1)+1)$-th to $(\frac{P}{N}*i)$-th partitions in the i-th round, for $1 \leq i \leq N$. In this simulation, a scheduler may pick slots from any partition in the cluster (i.e., using optimistic scheduling).

We vary task submission rate R, synchronization gap G, the variance of slot scores V, and the number of schedulers (i.e., AN) from N to $4N$, where $N = 10$. As shown in Figures 3(a)-(c), SameSyn requires more extra slots than DiffSyn in order to achieve the same latency, which can be explained as follows.

We found that schedulers tend to pick slots from fresher partitions. This is because when a scheduler tries to commit a task to a slot, it also updates the status of the slot in its local partition to "taken" (either by this task or already taken). Over time, more and more slots in the staler partitions are marked "taken", which are refreshed until the partitions are synchronized. This becomes a problem for SameSyn because all schedulers are competing for available slots from the same fresher partitions, thus leading to more conflicts. In contrast, under DiffSyn, the fresher partitions of each scheduler are different, meaning that each scheduler always has higher commit rate in its own fresher partitions. Each scheduler under DiffSyn still has high conflict rate when committing tasks to its staler partitions, but this situation also happens to SameSyn.

Scheduling strategies. We examine three scheduling strategies: **Quality-first**, **Latency-first**, and **Adaptive**. In Quality-first, a scheduler schedules a task by first choosing the partition with the highest average slot score and then picking available slots by weighted sampling based on slot scores. In Latency-first, each scheduler schedules a task by first picking slots from its freshest partitions, but if suitable slots are not found, then it looks for slots in other partitions (from the least stale partition first).

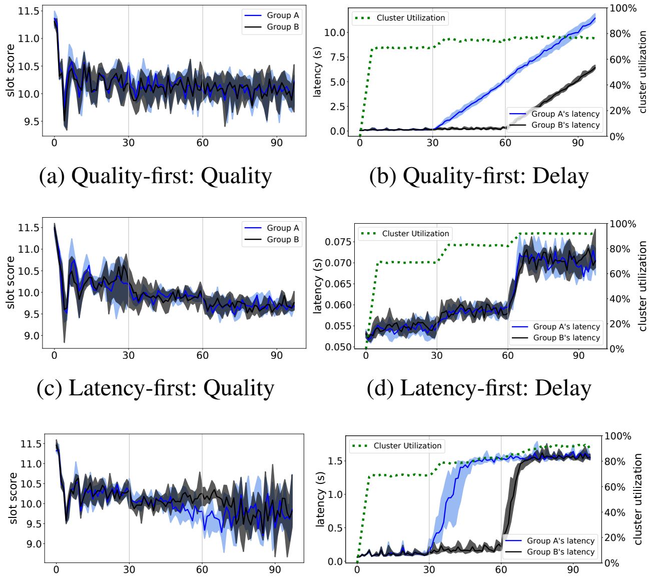
Adaptive uses Quality-first when it does not incur much scheduling delay, while it adopts Latency-first when scheduling efficiency is more critical. It calculates an exponential moving average (EMA) of scheduling delay, such that Quality-first is used when the EMA is smaller than a threshold τ, and Latency-first is used otherwise. Note that τ is usually the SLA for the scheduling delay specified by the cluster operator or users. Here we intend to show that the adaptive strategy can bound the scheduling delay by τ when the cluster is busy, and attain high slot quality as Quality-first when the cluster is not busy, instead of arguing what is the best trade-off we can make. We run simulation by setting $\tau = 1.5$s as a demonstration.

We simulate three scenarios that may happen in our production cluster on a typical day as follows. We create two groups of schedulers, A and B, and divide the timeline into three phases: (1) A and B are both operating at 2/3 of their full capacity; (2) A is operating at full capacity while B remains the same; (3) A and B are both operating at full capacity. Each phase is run for 30s.

Figure 4 plots the median and the 10th-90th interval of the slot scores and scheduling delay of Group A (dark color) and Group B (pale color), respectively. We also plot the cluster utilization rate as a thick dashed curve in Figures 4b, 4d and 4f for reference.

During Phase 1, the three strategies have similar slot quality and scheduling delay. This is because the task submission rate is only 2/3 of the full load and the cluster is only 60%-70% utilized, and thus all schedulers (from both groups) have high-quality slots to pick without incurring many conflicts.

During Phase 2, the three strategies start to behave differently as Group A is now undergoing a stress test. When Quality-first is used, Figure 4a shows that the slot quality is not degraded compared with Phase 1; but Figure 4b reveals the real problem as the scheduling delay of Group A schedulers surges, while that of Group B remains stable. Thus, Quality-first is not effective when the scheduling load is high. When Latency-first is used, Figures 4c&4d show that both the slot quality and scheduling delay of both groups only become slightly worse. This is because even though Group A is fully loaded, the overall cluster utilization rate is only about 80% and thus the schedulers can still have enough quality slots to pick from their freshest partitions. When Adaptive is used, Figure 4f shows that the scheduling delay of Group A



(a) Quality-first: Quality

(b) Quality-first: Delay

(c) Latency-first: Quality

(d) Latency-first: Delay

(e) Adaptive: Quality

(f) Adaptive: Delay

Figure 4: The performance of different scheduling strategies (dotted green line denotes the cluster utilization)

increases to 1.5s and then remains bounded there, as τ is set at 1.5s (note that this is much lower than the delay of Group A in Figure 4b, which rises to 4.7s). This means that Group A schedulers now switch to use Latency-first, while Group B still use Quality-first. As a result, Group A now has lower slot quality than Group B as shown in Figure 4e. Although Group B uses Quality-first while Group A uses Latency-first, the scheduling delay of Group B is actually much lower than that of Group A in Phase 2 of Figure 4f because Group B is only operating at 2/3 of its full capacity.

During Phase 3, even though Quality-first maintains the same slot quality, the scheduling delay of both groups keeps increasing. The slot quality of both Latency-first and Adaptive worsens compared with Phase 2 due to the heavier load and now the schedulers are competing more for quality slots. Figure 4d shows that the delay of Latency-first for both groups also increases, though still much lower than Quality-first and Adaptive. Adaptive bounds the delay of both groups at 1.5s.

In summary, Quality-first works well when the scheduling load or the cluster load is not heavy, as it leads to high-quality slot allocation. When the load is heavy, the delay of Quality-first may become too high and Latency-first is preferred as it achieves both high efficiency and slot quality. Adaptive is also a good choice as Adaptive keeps the latency bounded (which is particularly desirable as the bound can be tied to the SLA), while Adaptive can also take advantage of Quality-first to get high-quality slots when the load is not heavy.

To deploy ParSync in our cluster, we discuss some details to important issues such as what changes are needed in the upgrade, how the scheduling objectives are achieved, and how the cluster is partitioned in Supplemental Material B in [18] .

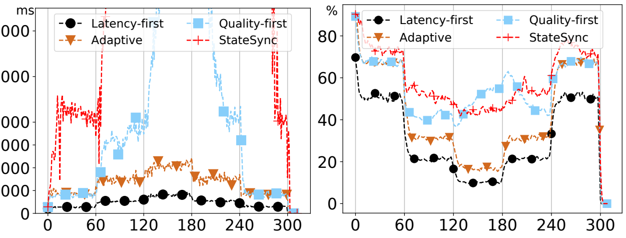
6 Performance Evaluation

We conducted our experiments in a high-fidelity testing environment called *Wind Tunnel*. Note that although ParSync is deployed in large production clusters in our company, conducting real experiments to evaluate ParSync's scalability on these large clusters is prohibited by company policy as not to cause unexpected interruption to normal business operations. The company relies heavily on Wind Tunnel to test the robustness and performance of every cluster scheduler component and scheduling algorithm, and any change must be tested extensively in Wind Tunnel before applied to the production clusters.

The results obtained from Wind Tunnel are close to the true performance of the schedulers in our production clusters as Wind Tunnel uses the same codebase of the entire production cluster system and it incorporates critical factors from our production environment. First, in Wind Tunnel, each scheduler or resource manager is deployed on an independent machine and executes code used in production. A set of machines is then used to simulate resource slots in a production cluster. The interaction between schedulers, resource managers, and workers are the same as in production. Wind Tunnel makes the staleness of local states to be more reflective of real situations by deploying resource managers and schedulers on a real network topology. Second, the schedulers execute the same scheduling logic as in our production cluster, which enables the evaluation to reproduce the true scheduling capacity of each scheduler. Third, each job has a list of *preferred slots* computed at runtime for its tasks (instead of using a static score for each slot as in the simulations in §5.2) to better reflect the contentions on slots in reality.

The main difference between the high-fidelity simulation by Wind Tunnel and a production cluster is that the execution of a task is effectively "sleeping", and each worker machine in Wind Tunnel is simulating many worker machines in a production cluster. All the rest are based on the same code base. Therefore, the results obtained from Wind Tunnel are similar to the results obtained from the production clusters.

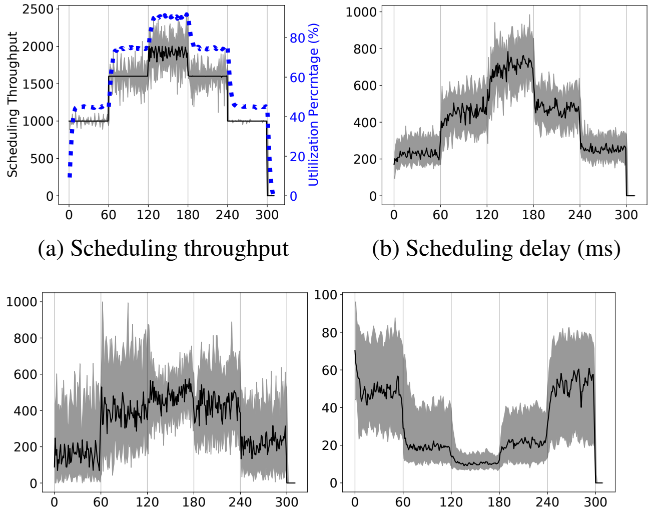
Settings. In the experiments, we used the following default settings unless otherwise specified. We deployed 20 machines in Wind Tunnel for schedulers and 2 machines for resources managers. We used another 30 machines in Wind Tunnel to simulate 200k slots and each resource manager oversees 100k slots. The workload we used was sampled from the production trace whose statistics is shown in Figure 1. The baseline peak task submission rate from the sampled workload is around 40k/s, and we varied the rate to 50%, 80% and 95% of 40k/s to simulate different levels of pressure. The cluster state was partitioned into 20 parts in ParSync. The synchronization gap in our production cluster is around 0.5s.



(a) Scheduling delay (ms)

(b) Scheduling quality (%)

Figure 5: Performance of ParSync and StateSync



(a) Scheduling throughput

(b) Scheduling delay (ms)

(c) Number of conflicts

(d) Scheduling quality (%)

Figure 6: Results of ParSync using Latency-first

Performance metrics. We use *scheduling delay and scheduling quality* as the major measures. Scheduling quality is the percentage of tasks that are allocated to their preferred slots. We also report the *scheduling throughput* and *the number of conflicts* to analyze the performance.

6.1 Performance Comparison

There are many works related to cluster scheduling (§3). The scheduling algorithms mentioned in 3.3 are orthogonal to our work as many of them can also be applied for scheduling in ParSync. As discussed in §3.2, most existing schedulers are not as general as Omega's shared-state design (§3.1), as they have specific requirements (e.g., accurate task duration estimation, strong disk locality) or cannot provide important features such as backward compatibility, support of multiple scheduling algorithms and global view. While these schedulers are effective in their own settings, we do not compare ParSync with them since the design objectives are different (and it is also difficult to create an environment in Wind Tunnel for a fair comparison, if possible, with these systems).

Instead, we compared ParSync with StateSync, which simulates Omega. We also tested StateSync with only 1 scheduler, which simulates our previous centralized scheduler (similar to YARN), but the delay is nearly two orders of magnitude worse than StateSync and thus we do not report the details.

For ParSync, we tested Latency-first, Quality-first and Adaptive (τ =2s) introduced in §5.2. StateSync adopts the shared-state architecture in §4, which synchronizes the entire state each time. Both ParSync and StateSync used 20 schedulers. Tasks were submitted during a 300-second period, which was divided into five 60-second phases with different submission rates: 50%, 80%, 95%, 80%, 50% of 40k/s. The 50%, 80% and 95% rates simulate medium, medium-heavy and heavy loads of a cluster.

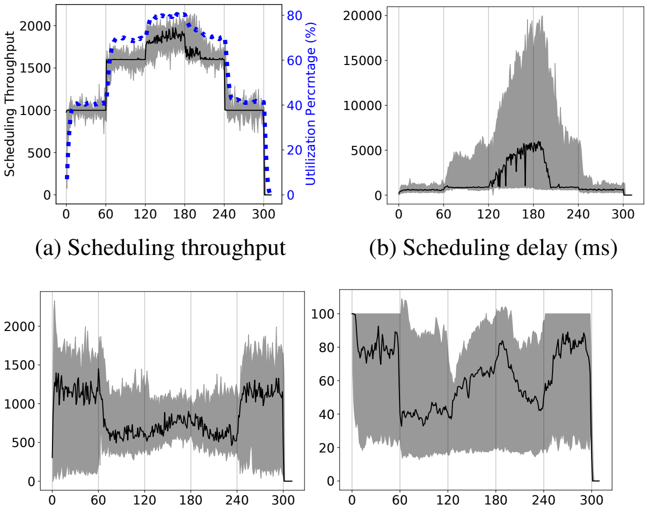
Figure 5a shows that the average scheduling delay of ParSync using Latency-first is significantly smaller than the other approaches throughout the five phases. The delay of ParSync using Adaptive is bounded by τ. The delay of Quality-first increases much faster than Latency-first and Adaptive during the peak periods (e.g., Phase 3), but is still much smaller than that of StateSync. Starting from Phase 2, StateSync's delay increases rapidly as its schedulers are not able to handle the high task submission rate and uncommitted tasks starts to accumulate. The high delay starts to drop only in Phase 5 when the accumulated tasks from the previous phases are gradually committed. Although Figure 5b shows that *the scheduling quality of StateSync is actually quite good, its high scheduling delay makes it infeasible for production use*.

The results verify our findings in §4 that *sharing the whole cluster state without partitioned synchronization cannot scale to handle high task submission rates in our cluster, as the number of conflicts and the scheduling delay increase rapidly*.

6.2 Scheduling Delay and Scheduling Quality

Next we analyze in details the performance of ParSync, as reported in Figure 6 and Figure 7. In each figure, the dark curve plots the median and the shadows plot the 10- and 90-percentile values among the 20 schedulers.

Scheduling throughput. Figures 6a and 7a show that both Latency-first and Quality-first can handle the task submission rate. In Phase 3, schedulers using Quality-first have throughput lower than 1.9K/s in Figure 7a (note that on average, each of the 20 schedulers receives 1K, 1.6K, 1.9K, 1.6K, and 1K tasks per second in each of the five phases). This is because of the surge in the scheduling delay in Phase 3 (Figure 7b), and hence some tasks are accumulated and only successfully committed during Phase 4. This also explains why the throughput of most schedulers goes above 1.6K/s at the beginning of Phase 4. Figures 6a and 7a also show that the



(a) Scheduling throughput

(b) Scheduling delay (ms)

(c) Number of conflicts

(d) Scheduling quality (%)

Figure 7: Results of ParSync using Quality-first

cluster utilization (plotted in thick dash curves) is reflected by the scheduling throughput in each phase.

Scheduling delay. Figure 6b shows that the scheduling delay of Latency-first increases with the task submission rate, which is mainly due to increasing scheduling conflicts as shown in Figure 6c. The median and mean delay roughly follow the task submission rate in each phase. We also observed that the total number of uncommitted tasks accumulated by each scheduler in each second is small and does not keep growing even during Phase 3 (see detailed results in Supplemental Material A.1 in [18]). This demonstrates that Latency-first is able to sustain and provide reasonable latency under heavy scheduling loads even if the peak period continues. In contrast, Figure 7b shows that using Quality-first, the scheduling delay increases rapidly in Phase 2 and Phase 3. The delay starts to drop in Phase 4 but is still high because it needs to clear the accumulated uncommitted tasks, which surges from 1,700 in Phase 2 to 5,800 in Phase 3 (1,700 and 5,800 are median values, details are reported in Supplemental Material A.1 in [18]). This is because Quality-first has a higher scheduling overhead as it checks all partitions to find preferred slots, its delay increases quickly when there are a large number of tasks to schedule. We also remark that Quality-first's high delay is not primarily due to conflicts, as the number of conflicts of Quality-first is not much larger than that of Latency-first in Phase 3 as shown in Figures 6b and 7c.

Number of Conflicts. Figure 6c shows that Latency-first has an insignificant number of conflicts in Phase 1 as there are plenty of idle slots inside the freshest partition of each scheduler. There are more conflicts during Phase 2 to Phase 4 as a scheduler may schedule some tasks to other less fresh partitions due to insufficient idle slots in its freshest partition. Interestingly, we observe an inverse pattern in Figure 7c, as Quality-first has a higher number of conflicts in Phase 1 and Phase 5 than in other phases. We examined the details and found that, in Phase 1 and Phase 5, Quality-first schedulers schedule tasks to preferred slots in all the partitions evenly since they all have plenty of idle slots. But from Phase 2, each scheduler starts to have an *increasingly* more idle slots in its own fresher partitions than in staler partitions (as discussed in §5.1). Thus, each scheduler tends to schedule more tasks to its fresher partitions, which incurs fewer conflicts.

Scheduling quality. Figures 6d and 7d show that Quality-first achieves considerably better quality than Latency-first. Note that there are always some tasks that may not get their preferred slots, as tasks are competing for preferred slots that may not be enough for all the tasks wanting them. Thus, the percentage of tasks getting their preferred slots drops accordingly when more tasks are submitted in Phases 2-4. However, Quality-first is actually able to allocate most slots to tasks that prefer them, although this is at the cost of higher delay when the scheduling load is heavy.

Compared with the more ideal simulation results in Figure 4, we observe more variations in the results shown in Figures 6 and 7 due to the introduction of real factors in our production cluster. But overall, the results are still consistent with our findings in §5.2, except that the trade-off between latency and quality becomes more obvious in Figures 6 and 7. The results in Figure 5 also validate that StateSync cannot handle our high task submission rates as ParSync does.

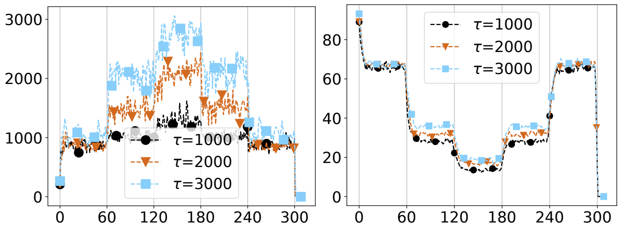
6.3 The Performance of the Adaptive Strategy

In this set of experiments, we evaluated how the latency threshold τ affects the performance of ParSync using the Adaptive strategy, by setting $\tau = 1000$, 2000 and 3000. The range $1000 \leq \tau \leq 3000$ is representative of our workloads and it is not common to set the delay bound higher than $3,000$ ms. We ran the same five phases as in the experiments in §6.2.

Figure 8 reports the median values (more details are reported in Supplemental Material A.2 in [18]) of the scheduling delay and quality of all schedulers. For all values of τ, Adaptive's performance in Phases 1 and 5 is similar to that of Quality-first (see Figure 7), because Adaptive uses Quality-first when the scheduling load is not heavy. From Phase 2 to Phase 4, while Quality-first's delay increases quickly as reported in Figure 7b, Figure 8a shows that Adaptive can effectively bound the delay at τ. Whenever the EMA is greater than τ during Phases 2-4, Adaptive switches to use Latency-first and thus it does not accumulate uncommitted tasks as in Quality-first. Consequently, its delay also drops quickly from Phase 3 to Phase 4, in contrast to Figure 7b. However,

Table 1: The effects of slot quality: in each cell, *left middle right* values = *Latency-first Adaptive(*τ=1000*) Quality-first* results (**STET**: average sum of task execution time; **JSD**: average job scheduling delay; **JCT:** average job completion time)

Metric(sec)/α	0.9	0.8	0.7	0.6	0.5	0.4
STET	922 914 904	888 873 854	858 822 772	836 788 712	797 726 622	762 600 542
JSD	0.4 0.8 3.7	0.4 0.8 3.3	0.4 0.8 2.9	0.4 0.8 2.0	0.4 0.8 1.9	0.4 0.7 1.5
JCT	10.3 10.6 12.7	10.3 10.5 12.1	10.2 10.3 11.2	10.2 10.0 9.9	10.1 9.5 9.1	10.1 8.6 8.4



(a) Scheduling delay (ms)

(b) Scheduling quality (%)

Figure 8: Results of the Adaptive Strategy

Figure 8b shows that Adaptive's scheduling quality also drops in Phases 2-4, although its quality is still higher than that of Latency-first in Figure 6d because Adaptive uses Quality-first to get more quality slots until the EMA reaches τ. This is also why the quality of τ = 3000 is generally better than those of τ = 2000 and τ = 1000.

Overall, *the results demonstrate the effectiveness of Adaptive in balancing scheduling delay and quality under different values of* τ. We report more details in Supplemental Material A.2, which show that we can choose different values of τ to bound scheduling delay without significantly affecting scheduling quality, throughput and the number of conflicts.

6.4 The Effects of Slot Quality

Running a task in its preferred slot reduces the task execution time (**TET**). The shorter the TET, the more room it creates for tolerating scheduling delay. Thus, if scheduling tasks to their preferred slots can shorten their TET significantly, Quality-first could be preferred to Latency-first. To test the effects of slot quality, we use a factor α to adjust the benefit of running a task in its preferred slot as follows: if the TET of running a task in a non-preferred slot is t, then its TET in a preferred slot is αt, where $0 < \alpha < 1$.

Table 1 reports the average sum of TET (**STET**, i.e., the sum of the TET of all tasks in a job, averaged over all jobs), average job scheduling delay (**JSD**, i.e., the duration between the time when a job is submitted and when all its tasks are committed), and average job completion time (**JCT**, i.e., the duration between the time when a job is submitted and when all its tasks are completed).

When we increase the benefit of preferred slots (i.e., smaller α), Quality-first is favored in terms of STET since it schedules more tasks to their preferred slots than both Latency-first and Adaptive as reported in §6.2 and §6.3. The JSD of Quality-first also decreases as α becomes smaller, because shorter TET releases occupied slots to other tasks earlier and hence leads to fewer conflicts. However, due to the higher scheduling overhead of Quality-first, its JSD is still much larger than that of Latency-first. In terms of JCT, since it benefits from both shorter TET and JSD, Quality-first starts to achieve a smaller JCT than Latency-first when the benefit of preferred slots exceeds its scheduling overhead, i.e., when $\alpha \leq 0.6$. In comparison, *the JCT of Adaptive always closely follows the best one of Quality-first and Latency-first, which proves the effectiveness of the adaptive strategy.*

6.5 Other Experimental Results

We also examined the effects of the number of partitions (which also changes the synchronization gap) and the scalability of ParSync. Due to the page limit, we report the details in Supplemental Material A.3 and A.4 in [18], and summarize the results here: (1) increasing the partition number has almost no impact on the scheduling performance; and (2) ParSync achieves stable performance as the scale of the cluster and task submission rate increases by 1 to 4 times of the capacity of our current cluster.

7 Conclusions

We presented ParSync, which increases the scheduling capacity of our production cluster from a few thousand tasks per second on thousands of machines to 40K tasks/sec on 100K machines. ParSync effectively reduces conflicts in contending resources to achieve low scheduling delay and high scheduling quality. The simplicity of ParSync allows us to maintain user transparency and backward compatibility that are essential to our production clusters.

Acknowledgments. We thank the reviewers for their constructive comments that have helped greatly improve the quality of the paper. This work was supported by the Alibaba-CUHK collaboration project "Large Scale Cluster Scheduling" (code: 7010574).

References

[1] 40 and 100 Gigabit Ethernet. http://www.extremenetworks.com.

[2] Apache Arrow - Powering Columnar In-Memory Analytics. https://arrow.apache.org/.

[3] Apache Avro. https://avro.apache.org/.

[4] Apache ORC - High-Performance Columnar Storage for Hadoop. https://orc.apache.org/.

[5] Apache Parquet. https://parquet.apache.org/.

[6] Technologies for Data-Intensive Computing. http://www.hpts.ws/agenda.html.

[7] Ganesh Ananthanarayanan, Ali Ghodsi, Scott Shenker, and Ion Stoica. Disk-locality in datacenter computing considered irrelevant. In *HotOS*, volume 13, pages 12–12, 2011.

[8] Ganesh Ananthanarayanan, Ali Ghodsi, Scott Shenker, and Ion Stoica. Effective straggler mitigation: Attack of the clones. In Nick Feamster and Jeffrey C. Mogul, editors, *Proceedings of the 10th USENIX Symposium on Networked Systems Design and Implementation, NSDI 2013, Lombard, IL, USA, April 2-5, 2013*, pages 185–198. USENIX Association, 2013.

[9] Eric Boutin, Jaliya Ekanayake, Wei Lin, Bing Shi, Jingren Zhou, Zhengping Qian, Ming Wu, and Lidong Zhou. Apollo: Scalable and coordinated scheduling for cloud-scale computing. In Jason Flinn and Hank Levy, editors, *11th USENIX Symposium on Operating Systems Design and Implementation, OSDI '14, Broomfield, CO, USA, October 6-8, 2014*, pages 285–300. USENIX Association, 2014.

[10] Mosharaf Chowdhury, Zhenhua Liu, Ali Ghodsi, and Ion Stoica. Hug: Multi-resource fairness for correlated and elastic demands. In *Proceedings of the 13th USENIX Symposium on Networked Systems Design and Implementation*, pages 407–424, 2016.

[11] Emilio Coppa and Irene Finocchi. On data skewness, stragglers, and mapreduce progress indicators. In *Proceedings of the Sixth ACM Symposium on Cloud Computing*, pages 139–152. ACM, 2015.

[12] Eli Cortez, Anand Bonde, Alexandre Muzio, Mark Russinovich, Marcus Fontoura, and Ricardo Bianchini. Resource central: Understanding and predicting workloads for improved resource management in large cloud platforms. In *Proceedings of the 26th ACM Symposium on Operating Systems Principles*, pages 153–167. ACM, 2017.

[13] Pamela Delgado, Diego Didona, Florin Dinu, and Willy Zwaenepoel. Job-aware scheduling in Eagle: Divide and stick to your probes. In *Proceedings of the Seventh ACM Symposium on Cloud Computing*, pages 497–509. ACM, 2016.

[14] Pamela Delgado, Florin Dinu, Anne-Marie Kermarrec, and Willy Zwaenepoel. Hawk: Hybrid datacenter scheduling. In Shan Lu and Erik Riedel, editors, *2015 USENIX Annual Technical Conference, USENIX ATC '15, July 8-10, Santa Clara, CA, USA*, pages 499–510. USENIX Association, 2015.

[15] Christina Delimitrou and Christos Kozyrakis. Paragon: QoS-aware scheduling for heterogeneous datacenters. In *Proceedings of the 18th International Conference on Architectural Support for Programming Languages and Operating Systems*, volume 48, pages 77–88. ACM, 2013.

[16] Christina Delimitrou and Christos Kozyrakis. Quasar: resource-efficient and QoS-aware cluster management. In *Proceedings of the 19th International Conference on Architectural Support for Programming Languages and Operating Systems*, volume 42, pages 127–144. ACM, 2014.

[17] Christina Delimitrou, Daniel Sanchez, and Christos Kozyrakis. Tarcil: reconciling scheduling speed and quality in large shared clusters. In *Proceedings of the 6th ACM Symposium on Cloud Computing*, pages 97–110. ACM, 2015.

[18] Yihui Feng, Zhi Liu, Yunjian Zhao, Tatiana Jin, Yidi Wu, Yang Zhang, James Cheng, Chao Li, and Tao Guan. Scaling large production clusters with partitioned synchronization. In *Technical Report (http://www.cse.cuhk.edu.hk/~jcheng/papers/parsync_atc21.pdf)*. CUHK, 2021.

[19] Ali Ghodsi, Matei Zaharia, Benjamin Hindman, Andy Konwinski, Scott Shenker, and Ion Stoica. Dominant resource fairness: Fair allocation of multiple resource types. In *Proceedings of the 8th USENIX Symposium on Networked Systems Design and Implementation*, volume 11, pages 24–24, 2011.

[20] Ionel Gog, Malte Schwarzkopf, Adam Gleave, Robert NM Watson, and Steven Hand. Firmament: Fast, centralized cluster scheduling at scale. In *Proceedings of the 12th USENIX Symposium on Operating Systems Design and Implementation*, pages 99–115, 2016.

[21] Robert Grandl, Ganesh Ananthanarayanan, Srikanth Kandula, Sriram Rao, and Aditya Akella. Multi-resource packing for cluster schedulers. In *Proceedings of the ACM SIGCOMM Computer Communication Review*, volume 44, pages 455–466. ACM, 2015.

[22] Robert Grandl, Mosharaf Chowdhury, Aditya Akella, and Ganesh Ananthanarayanan. Altruistic scheduling in multi-resource clusters. In *Proceedings of the 12th USENIX Symposium on Operating Systems Design and Implementation*, pages 65–80, 2016.

[23] Robert Grandl, Srikanth Kandula, Sriram Rao, Aditya Akella, and Janardhan Kulkarni. Graphene: Packing and dependency-aware scheduling for data-parallel clusters. In *Proceedings of the 12th USENIX Symposium on Operating Systems Design and Implementation*, pages 81–97, 2016.

[24] Benjamin Hindman, Andy Konwinski, Matei Zaharia, Ali Ghodsi, Anthony D. Joseph, Randy H. Katz, Scott Shenker, and on Stoica. Mesos: A platform for fine-grained resource sharing in the data center. In David G. Andersen and Sylvia Ratnasamy, editors, *Proceedings of the 8th USENIX Symposium on Networked Systems Design and Implementation, NSDI 2011, Boston, MA, USA, March 30 - April 1, 2011*. USENIX Association, 2011.

[25] Benjamin Hindman, Andy Konwinski, Matei Zaharia, Ali Ghodsi, Anthony D Joseph, Randy H Katz, Scott Shenker, and Ion Stoica. 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*, volume 11, pages 22–22, 2011.

[26] Yuzhen Huang, Yingjie Shi, Zheng Zhong, Yihui Feng, James Cheng, Jiwei Li, Haochuan Fan, Chao Li, Tao Guan, and Jingren Zhou. Yugong: Geo-distributed data and job placement at scale. *Proc. VLDB Endow.*, 12(12):2155–2169, 2019.

[27] Michael Isard, Vijayan Prabhakaran, Jon Currey, Udi Wieder, Kunal Talwar, and Andrew Goldberg. Quincy: fair scheduling for distributed computing clusters. In *Proceedings of the 22nd ACM Symposium on Operating Systems Principles*, pages 261–276. ACM, 2009.

[28] Tatiana Jin, Zhenkun Cai, Boyang Li, Chengguang Zheng, Guanxian Jiang, and James Cheng. Improving resource utilization by timely fine-grained scheduling. In Angelos Bilas, Kostas Magoutis, Evangelos P. Markatos, Dejan Kostic, and Margo I. Seltzer, editors, *EuroSys '20: Fifteenth EuroSys Conference 2020, Heraklion, Greece, April 27-30, 2020*, pages 20:1–20:16. ACM, 2020.

[29] Prajakta Kalmegh and Shivnath Babu. Mifo: A query-semantic aware resource allocation policy. In *Proceedings of the 2019 ACM International Conference on Management of Data*, pages 1678–1695. ACM, 2019.

[30] Konstantinos Karanasos, Sriram Rao, Carlo Curino, Chris Douglas, Kishore Chaliparambil, Giovanni Matteo Fumarola, Solom Heddaya, Raghu Ramakrishnan, and Sarvesh Sakalanaga. Mercury: Hybrid centralized and distributed scheduling in large shared clusters. In *2015 USENIX Annual Technical Conference*, pages 485–497, 2015.

[31] James E Kelley Jr and Morgan R Walker. Critical-path planning and scheduling. In *Papers presented at the December 1-3, 1959, eastern joint IRE-AIEE-ACM computer conference*, pages 160–173. ACM, 1959.

[32] Kubernetes. Taking network bandwidth into account for scheduling. https://github.com/kubernetes/kubernetes/issues/16837, 2015.

[33] YongChul Kwon, Magdalena Balazinska, Bill Howe, and Jerome Rolia. Skewtune: mitigating skew in mapreduce applications. In *Proceedings of the 2012 ACM SIGMOD International Conference on Management of Data*, pages 25–36. ACM, 2012.

[34] Libin Liu and Hong Xu. Elasecutor: Elastic executor scheduling in data analytics systems. In *Proceedings of the ACM Symposium on Cloud Computing*, pages 107–120. ACM, 2018.

[35] Kay Ousterhout, Aurojit Panda, Josh Rosen, Shivaram Venkataraman, Reynold Xin, Sylvia Ratnasamy, Scott Shenker, and Ion Stoica. The case for tiny tasks in compute clusters. In Petros Maniatis, editor, *14th Workshop on Hot Topics in Operating Systems, HotOS XIV, Santa Ana Pueblo, New Mexico, USA, May 13-15, 2013*. USENIX Association, 2013.

[36] Kay Ousterhout, Patrick Wendell, Matei Zaharia, and Ion Stoica. Sparrow: Distributed, low latency scheduling. In *Proceedings of the Twenty-Fourth ACM Symposium on Operating Systems Principles*, pages 69–84. ACM, 2013.

[37] Jeff Rasley, Konstantinos Karanasos, Srikanth Kandula, Rodrigo Fonseca, Milan Vojnovic, and Sriram Rao. Efficient queue management for cluster scheduling. In Cristian Cadar, Peter R. Pietzuch, Kimberly Keeton, and Rodrigo Rodrigues, editors, *Proceedings of the Eleventh European Conference on Computer Systems, EuroSys 2016, London, United Kingdom, April 18-21, 2016*, pages 36:1–36:15. ACM, 2016.

[38] Krzysztof Rzadca, Pawel Findeisen, Jacek Swiderski, Przemyslaw Zych, Przemyslaw Broniek, Jarek Kusmierek, Pawel Nowak, Beata Strack, Piotr Witusowski, Steven Hand, and John Wilkes. Autopilot: workload autoscaling at google. In Angelos Bilas, Kostas Magoutis, Evangelos P. Markatos, Dejan Kostic, and Margo Seltzer,

editors, *EuroSys '20: Fifteenth EuroSys Conference 2020, Heraklion, Greece, April 27-30, 2020*, pages 16:1–16:16. ACM, 2020.

[39] Malte Schwarzkopf, Andy Konwinski, Michael Abd-El-Malek, and John Wilkes. Omega: Flexible, scalable schedulers for large compute clusters. In Zdenek Hanzálek, Hermann Härtig, Miguel Castro, and M. Frans Kaashoek, editors, *Eighth Eurosys Conference 2013, EuroSys '13, Prague, Czech Republic, April 14-17, 2013*, pages 351–364. ACM, 2013.

[40] Xiaoyang Sun, Chunming Hu, Renyu Yang, Peter Garraghan, Tianyu Wo, Jie Xu, Jianyong Zhu, and Chao Li. Rose: Cluster resource scheduling via speculative over-subscription. In *2018 IEEE 38th International Conference on Distributed Computing Systems*, pages 949–960. IEEE, 2018.

[41] Muhammad Tirmazi, Adam Barker, Nan Deng, Md E. Haque, Zhijing Gene Qin, Steven Hand, Mor Harchol-Balter, and John Wilkes. Borg: the next generation. In Angelos Bilas, Kostas Magoutis, Evangelos P. Markatos, Dejan Kostic, and Margo Seltzer, editors, *EuroSys '20: Fifteenth EuroSys Conference 2020, Heraklion, Greece, April 27-30, 2020*, pages 30:1–30:14. ACM, 2020.

[42] Animesh Trivedi, Patrick Stuedi, Jonas Pfefferle, Adrian Schüpbach, and Bernard Metzler. Albis: High-performance file format for big data systems. In Haryadi S. Gunawi and Benjamin Reed, editors, *2018 USENIX Annual Technical Conference, USENIX ATC 2018, Boston, MA, USA, July 11-13, 2018*, pages 615–630. USENIX Association, 2018.

[43] Alexey Tumanov, Timothy Zhu, Jun Woo Park, Michael A Kozuch, Mor Harchol-Balter, and Gregory R Ganger. Tetrisched: global rescheduling with adaptive plan-ahead in dynamic heterogeneous clusters. In *Proceedings of the 11th European Conference on Computer Systems*, page 35. ACM, 2016.

[44] Vinod Kumar Vavilapalli, Arun C Murthy, Chris Douglas, Sharad Agarwal, Mahadev Konar, Robert Evans, Thomas Graves, Jason Lowe, Hitesh Shah, Siddharth Seth, Saha Bikas, Carlo Curino, Owen O'Malley, Sanjay Radia, Benjamin Reed, and Eric Baldeschwieler. Apache Hadoop YARN: Yet another resource negotiator. In *Proceedings of the 4th annual Symposium on Cloud Computing*, page 5. ACM, 2013.

[45] Abhishek Verma, Luis Pedrosa, Madhukar Korupolu, David Oppenheimer, Eric Tune, and John Wilkes. Large-scale cluster management at Google with Borg. In Laurent Réveillère, Tim Harris, and Maurice Herlihy, editors, *Proceedings of the Tenth European Conference on Computer Systems, EuroSys 2015, Bordeaux, France, April 21-24, 2015*, pages 18:1–18:17. ACM, 2015.

[46] Yi Yao, Han Gao, Jiayin Wang, Ningfang Mi, and Bo Sheng. Opera: opportunistic and efficient resource allocation in Hadoop YARN by harnessing idle resources. In *Proceedings of the 25th International Conference on Computer Communication and Networks*, pages 1–9. IEEE, 2016.

[47] 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 *Proceedings of the 5th European conference on Computer systems*, pages 265–278. ACM, 2010.