

Megha: Decentralized Global Fair Scheduling for Federated Clusters

Meghana Thiyyakat

PES University

Bangalore, India

meghanathiyyakat@pesu.pes.edu

Subramaniam Kalambur

PES University

Bangalore, India

subramaniamkv@pes.edu

Dinkar Sitaram

Cloud Computing Innovation Council of India

dinkar@ccici.in

Abstract—Increasing scale and heterogeneity in data centers have led to the development of federated clusters such as KubeFed, Hydra, and Pigeon, that federate individual data center clusters. In our work, we introduce Megha, a novel decentralized resource management framework for such federated clusters. Megha employs flexible logical partitioning of clusters to distribute its scheduling load, ensuring that the requirements of the workload are satisfied with very low scheduling overheads. It uses a distributed global scheduler that does not rely on a centralized data store but, instead, works with eventual consistency, unlike other schedulers that use a tiered architecture or rely on centralized databases. Our experiments with Megha show that it can schedule tasks taking into account fairness and placement constraints with low resource allocation times - in the order of tens of milliseconds.

Index Terms—scheduling, resource management, federated architectures, fairness

I. INTRODUCTION

To meet the growing demands of scale, organizations are investing in data centers with hundreds of thousands of machines. These data centers are further divided into clusters based on application or utility. High utilization of this infrastructure evaluates to higher returns on investment. To achieve high utilization, the resources in a data center must be presented as a shareable unified pool to its users, and fine-grained allocation from this pool must be supported.

Due to the heterogeneity and complexity prevalent in recent workloads, scheduling and resource allocation are non-trivial problems. To manage a large number of worker nodes efficiently, scheduling frameworks are moving towards federated architectures wherein a top-level scheduler makes high-level scheduling decisions which are enforced by independent cluster schedulers. Hydra [6] and Pigeon[21] are two such schedulers that have embraced the federated architecture.

In our paper, we introduce Megha, a decentralized global scheduler that uses eventual consistency and flexible partitioning to overcome the limitations of existing schedulers. Megha has a federated architecture and divides the data center into smaller clusters, each of which is controlled by a Local Master (LM). It achieves high scheduling throughput by distributing the load to multiple top-level Global Masters (GMs). Each GM uses an eventually-consistent global view of the system's resources to make scheduling decisions. The LMs validate the decisions of the GM and deploy the tasks on the worker

nodes of the clusters handled by them. In our work, we also demonstrate how global fairness policies can be enforced with eventual consistency.

Our experiments were conducted with real-world production traces to evaluate Megha's task allocation speed and compare it with centralized and distributed scheduling approaches. The results show that Megha can achieve a median allocation time comparable to that of Sparrow [16], a random-sampling based distributed scheduler. Megha also reports two orders of magnitude better 99th percentile allocation times than Sparrow. The experiments also demonstrate how task placement constraints can be satisfied with eventually consistent global state while reporting lower allocation times than the centralized approach.

Our Contributions Our main contributions are listed below. We:

- define allocation time and break it down into its multiple components. We also analyze the contribution of each component towards allocation time in different scheduling approaches.
- introduce the design of a novel decentralized framework that uses eventual consistency and dynamic partitioning to overcome the limitations of existing approaches.
- explain how fairness can be implemented in a decentralized scheduler.
- demonstrate the improvement in allocation time of tasks scheduled using our approach when compared to a centralized scheduler, and a distributed scheduler(Sparrow).
- explore the behavior of the framework under different parameter settings.

The rest of the paper is organized as follows. Section II discusses related work. Section III defines allocation time and discusses its different components. Section IV gives an overview of the architecture of the framework and the fairness algorithm used. Section V describes the evaluation methodology. Section VI presents the results of the experiments. Section VII concludes the findings of the paper and Section VIII briefly describes future enhancements to our work.

II. RELATED WORK

A. Heterogeneity in workloads

To achieve better utilization of data center resources, jobs from multiple applications are co-located on the same data

center. Recent studies [5, 13, 17] have analysed production workload traces published by companies such as Google [22] and Alibaba[1]. An analysis[15] of the Alibaba cluster trace published in 2017 exposes the imbalance in the CPU and memory consumed by the tasks in the trace. This variation in the proportions in which the resource types are consumed advocates the need for fine-grained, independent resource allocation of each resource type. According to a study [11] of the cluster trace published in 2018, Alibaba’s workload consists of jobs from over 9K online services. Each online service has a different SLO in terms of tail latency. The study also discusses the variation in the characteristics of batch jobs and service jobs in terms of resources consumed and the runtime duration. Apart from the heterogeneity present within a workload, workloads also vary across datacenters. Due to this variation, it is recommended [2] that multiple cluster traces be used while evaluating scheduling solutions. In our work, we use three production traces to evaluate Megha.

B. Placement Constraints

From routine maintenance and upgrades in the data center, applications tend to run on heterogeneous hardware spanning multiple generations, running different software versions. Due to the diversity in the infrastructure, tasks have placement constraints that limit the machines on which the tasks can be run. Resource constraints dictate task placement based on resource availability, whereas placement constraints are based on the software and hardware configurations of machines, such as the OS kernel version and the clock speed of the CPU. In their work, Sharma et al. [19], discuss the impact of placement constraints on scheduling. According to the authors, taking into account placement constraints increases the delay in scheduling tasks by a factor of 2 to 6. However, these constraints cannot be ignored as they play an important role in ensuring optimal job performance. Nearly 50% of all tasks in Google’s workloads have simple non-combinatorial constraints. The results of our experiments with Megha are consistent with their findings that the scheduling overhead is very small when the load on the system is low. However, when the load increases, the unavailability of the resources that satisfy the tasks’ constraints translates to longer task wait times, and hence, higher allocation times.

C. Scheduling Approaches

Schedulers can be broadly classified into centralized, distributed, hybrid, and hierarchical schedulers. Centralized schedulers, such as YARN [20], can make optimal decisions because they have a global view of the system. However, as previous studies have shown [6] as the size of the data center increases, the processing and collection of large amounts of state lead to the formation of a bottleneck due to which these schedulers are unable to achieve the required scheduling throughput.

Distributed architectures such as Sparrow [16] rely on probability to find machines with available resources. For each task, Sparrow performs random sampling and inserts

probes into the worker queues to pick the machine with the lowest queuing time. The scheduling overhead in this approach is insignificant. But, when the data center is highly loaded, the probability of picking an available machine is very low. Therefore, distributed schedulers that rely on random sampling perform poorly under high utilization conditions.

The architecture that most closely resembles Megha’s is that of Pigeon [21]. Pigeon divides the data center into smaller groups, each of which is managed by a master. Distributed schedulers assign tasks in a job evenly across masters. In Megha, the GM is analogous to the distributed scheduler and the LM is analogous to the master. However, there are three main differences between the two architectures. Firstly, Pigeon does not perform fine-grained resource allocation. It divides its workers into fixed resource encapsulations called slots. Such coarse-grained allocation will lead to fragmentation and wastage of resources. Secondly, Pigeon uses weighted fair queuing and reservation to avoid long job starvation and to ensure low short job latency, respectively. Megha does not differentiate between job types. However, due to its decentralized nature, Megha achieves low allocation times and ensures against head-of-line blocking and starvation of any class of jobs. Thirdly, the groups in Pigeon once formed are fixed, that is, there is no scope for flexibility in the partitioning scheme. Megha allows tasks to be scheduled anywhere in the data center by using flexible partitioning. This allows better utilization of the data center’s resources, especially in the presence of placement constraints which restricts the number of eligible machines on which a task can run.

Hydra [6] is a federated resource management framework, created by Microsoft as Apollo’s [3] successor, to support the high scheduling decisions per second required by the workloads deployed on their data centers. Hydra divides its large cluster into smaller YARN sub-clusters. Each sub-cluster, therefore, has a Resource Manager (RM), that decides the task placement and performs the allocation of resources. Hydra introduces a component, called the AM-RM Proxy that allows tasks to span multiple sub-clusters. In Megha, the Global Master is analogous to the AM-RM Proxy while the Local Masters plays the role of the RMs. However, because each Megha GM maintains a (possibly stale) local copy of the global view of the entire system, it doesn’t need to contact the LMs every time a task request arrives. Hence, Megha can achieve lower median task allocation time (in the order of tens of milliseconds), than those reported by Hydra (2-3 seconds).

Hybrid schedulers such as Mercury [14], Hawk [7], and Eagle [8] use two sets of schedulers - a centralized scheduler for tasks that need resource guarantees and distributed schedulers for latency-sensitive tasks. These schedulers, however, suffer from the same problems as the distributed schedulers mentioned earlier. Moreover, due to the lack of coordination between the two sets of schedulers, global fairness policies cannot be implemented.

PCSSampler [12] extends the random sampling-based approach by caching state information received in response to probes. The usefulness of the cached state is highly dependent

on the workers probed. In large clusters under high loads, finding a worker with available slots may require multiple rounds of probing leading to high scheduling overheads. PCSSampler also does not take into account fairness. Megha's GMs, on the other hand, collect partial state information about an entire LM cluster after every launch request. The GMs also update their stored global state using periodic heartbeats from LMs. Megha ensures all its decisions are optimal by having the LMs validate them.

Other works [9, 10, 18, 23] optimize the performance of tasks by using machine learning to improve the task placement quality taking into account resource and task heterogeneity and the interference posed by co-located tasks.

III. ALLOCATION TIME

Short jobs from user-facing applications such as web searches and other queries typically run for hundreds of milliseconds. Since the quality of the users' experience depends on the response time of these jobs, they must be scheduled with negligible overheads. Such latency-sensitive tasks cannot tolerate resource allocation times greater than tens of milliseconds [16].

We define resource allocation time as the time it takes for a task to be allocated resources in a worker node and begin its execution. We measure resource allocation time as:

$$Allocation_Time = Task_Start - Task_Arrival_Time \quad (1)$$

$Task_Start$ is the time at which the task starts executing on a worker node and $Task_Arrival_Time$ is the time at which the task is inserted into the framework's request queue. Resource Allocation Time can be split further into 3 components:

$$Allocation_Time = Framework_Queuing_Delay + Processing_Delay + Worker_Queuing_Delay + Communication_Delay \quad (2)$$

$Framework_Queuing_Delay$ is the time the task spends in the framework's request queue waiting for processing. When the scheduling throughput of the framework is low, this delay can contribute to large resource allocation times. $Processing_Delay$ is the time taken by the framework to find a worker node that satisfies the requirements of a task. The framework must take into account resource constraints, and additionally, placement constraints, while finding an eligible worker node. Centralized scheduling frameworks need to process the entire global state of the system to find a worker node for a task. Simultaneously, they also need to ensure their state is kept up-to-date. While the consistent global view ensures that the scheduling decisions made are optimal, the increase in the number of workers leads to an increase in the amount of processing required, and consequently the creation of a bottleneck resulting in higher framework queuing delays. $Worker_Queuing_Delay$ is the time spent by the task in the chosen worker node's queue while it waits for resources to be allocated to it. Distributed scheduling

frameworks like Sparrow [16] and the distributed schedulers in hybrid frameworks such as Eagle[8], have an insignificant $Framework_Queuing_Delay$ component due to the high throughput of the frameworks and small $Processing_Delay$ component characteristic of random sampling. However, in these frameworks, the $Worker_Queuing_Delay$ component significantly impacts the $Allocation_Time$. Distributed schedulers do not have access to a global view of the system. Therefore, when the system is under high load, the schedulers are unable to find available worker nodes with certainty. This results in the queuing up of tasks at busy worker nodes even while there are available worker nodes in the system.

$Communication_Delay$ is the delay introduced due to communication between multiple components in the framework requiring them to send messages over the internal network. This value is dictated by the network speed of the data center and the number and size of the messages sent.

IV. MEGHA'S ARCHITECTURE

A. Overview

Megha has a decentralized architecture that combines the superior placement quality of centralized scheduling, with the scalability and low allocation times associated with distributed scheduling. The framework federates multiple smaller clusters in the data center and manages them as a single pool of resources. Each smaller cluster is governed by a Local Master (LM) which is analogous to a centralized resource manager such as YARN. The Local Master performs the typical functions of a resource manager- sends tasks to worker nodes in the cluster, checks for worker heartbeats, collects status information from workers, handles failure recovery of workers, etc. The task placement decision, of which node to launch a task on, are made by Global Masters (GMs) and the task launch requests are sent to the LMs. A depiction of Megha's architecture is shown in Fig. 1.

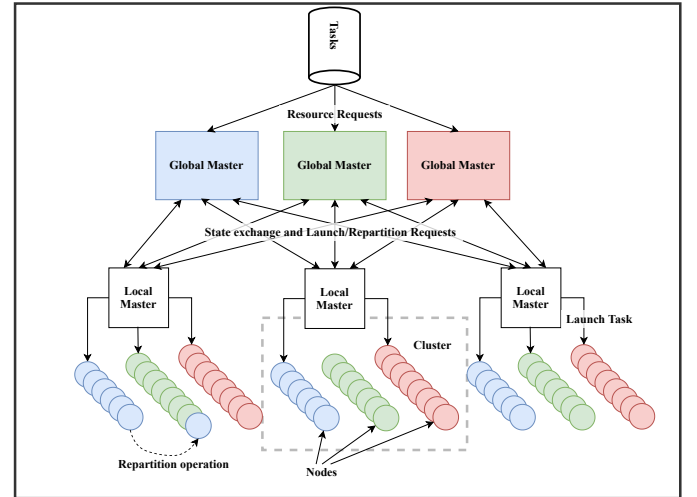


Fig. 1. Megha's architecture with one user queue

The GMs are independent of each other and do not communicate with one another. The sharing of the resources of the data center amongst them is facilitated by the LMs. Each LM periodically sends the GMs updates on the state of its clusters, enabling the GMs to have a global view of the entire system. However, the global state information maintained by a GM may be stale. The LMs on other hand have a partial view of the system - restricted to their cluster only; but, the state stored by an LM is always consistent and current. Therefore, the LMs are responsible for verifying if the GMs' requests can be satisfied by the true current state of the system. This is important because the resources requested by a GM may have been allocated to another GM. Aside from the periodic updates, the LMs also send updates to the GMs, piggybacking the response to task launch requests made by the GMs.

Each cluster under an LM is divided into logical sub-clusters called *partitions*. The LM assigns each of the GMs one of its partitions. Hence,

$$\text{count}(\text{partitions}) = \text{count}(\text{GMs}) \quad (3)$$

In Fig. 1, worker nodes in the cluster have been depicted using circles. The worker nodes in a partition have been colored uniformly, and the worker nodes are assigned to the correspondingly colored Global Master. Partitions assigned to a GM are known as its internal partitions. The remaining partitions, which belong to other GMs, are referred to as external partitions. Therefore, each GM has at its disposal the resources of a logical cluster formed by the aggregation of its internal partitions under each of the LMs. That is,

$$\text{GM_cluster}_j = \bigcup_{i=1}^{i=n} P_{ij} \quad (4)$$

where n is the number of LMs, and P_{ij} is the partition in LM_i assigned to the GM_j .

B. Tasks and Machine Constraints

We borrow the concept of placement constraints from previous work [19]. A machine is said to have a machine constraint if it has the property that satisfies the requirement denoted by the constraint. For example, a machine with a clock speed of 3.1 GHz satisfies the constraint "clock speed greater than 2.5GHz". A task placement constraint, on the other hand, denotes a preference or requirement of the task in terms of the hardware or software configuration of the machine. Therefore, a task, with the task placement constraint mentioned in the example above, can be scheduled on any machine with a clock speed greater than 2.5 GHz. A task can have zero or more placement constraints. A machine that satisfies all the placement constraints of a task is a potential candidate for the task to be scheduled on.

1) *Representation*: Each placement constraint is represented by a unique number in Megha. For every partition, the GM maintains a bit-vector per constraint such that each bit in the vector corresponds to a worker node in the partition. Therefore, the length of the bit-vector is equal to the number

of worker nodes in the partition. The value of the bit represents whether the corresponding node satisfies the requirements of the constraint. In our work, we have considered 21 task placement constraints. Thus, a task or machine can have any combination of the 21 constraints. The GM maintains 21 bit-vectors for every partition in the system - both internal and external.

2) *Processing Task Requests*: A task request, $T = (R, PC)$, made to a GM consists of the resource requirements of the task $R = (R_1, R_2 \dots R_n)$, and a list of task placement constraints $PC = (PC_1, PC_2 \dots PC_k)$. Here, n is the number of resource types, such that, $1 \leq n \leq r$, where r is the total number of resource types that Megha is aware of and R_i represents the number of units of resource i that the task requires. k is the number of task placement constraints, such that, $0 \leq k \leq m$, where m is the number of machine constraints available in the system.

A worker node in the system is of the form $N = (K, M)$, where K is the representation of the resources available in the worker node and M is the list of task placement constraints that the worker node satisfies, known as its machine constraints. On receiving a task request, the GM, uses its global state and Megha's matching algorithm, to pick a worker node from one of its internal partitions based on the task's placement constraints. That is, if a task has a list of placement constraints PC , such that $PC = (PC_1, PC_2 \dots PC_k)$, the GM searches for a worker node with machine constraints $M = (M_1, M_2 \dots M_l)$, such that

$$M \supseteq PC \quad (5)$$

where, a machine constraint $M_i \in M$, satisfies the task's placement constraint $PC_i \in PC$. The GM then checks its global state to see if the worker node has sufficient resources available. Therefore, if $K = (K_1, K_2 \dots K_j)$ is the list of resources available in the worker node N , where K_i denotes the number of units of resource i , and $R = (R_1, R_2 \dots R_n)$ is the resource requirement of the task T , then N is considered an eligible match for T if

$$\forall i \in n : K_i \geq R_i \quad (6)$$

3) *Match operation*: When a task request arrives at a GM, it chooses a partition from its internal partitions in a round-robin fashion. It then checks which worker nodes in the partition can satisfy the combination of placement constraints that the task requires. It does this by performing a bit-wise AND operation between all the bit-vectors corresponding to the task's constraints. The position of the positive bits in the resulting vector tells the GM which worker nodes in the partition satisfy all the placement constraints of the task. The GM then loops through the resource availability of the matching worker nodes and checks if any of the worker nodes can satisfy the resource requirements of the task. The match operation's algorithm has been shown in Algorithm 1. The GM sends the LM a launch request of the form (N, T) when it encounters a worker node with sufficient resources. The

Algorithm 1: Pseudo-code for the matching algorithm

Input: task request: $request$
task placement constraints: $constraints^T$
constraints matrix:
 $constraints^N[NUM_CONSTRAINTS]$

begin
//initialize to constraint vector of 1s
// size NUM_CONSTRAINTS
 $candidates = [11...1]$
Function $match_task(request, constraints^T)$
// bitwise AND of constraint vectors
foreach $constraint \in constraints^T$ **do**
 $candidates =$
 $candidates \text{ AND } constraints^N[constraint]$
end
foreach $candidate \in candidates$ **do**
 if $candidate == 1$ **AND**
 $request \leq candidate.resources$ **then**
 // match found successfully
 return $candidate$
 end
// no match found
return -1
end

LM confirms the availability of the requested resources and launches the task on the chosen worker node.

Since the match operation uses a bit representation, its time complexity can be represented as $\mathcal{O}(n.partition_size)$, where n is the number of task placement constraints. However, since the AND operation is performed either at the word or byte level, depending on the operand size in the CPU architecture, $\mathcal{O}(n.log_{size}partition_size)$, would be a more accurate representation of the time complexity, where $size$ denotes the operand size.

4) *Repartition operation:* When a task request cannot be satisfied by the internal partitions of a GM, the GM looks at the resources in the external partitions. If it finds an eligible worker node for the task, that is, a worker node that satisfies (5) and (6), it requests the LM to perform a repartition operation. In a repartition operation, the LM temporarily adds the required resources from the eligible worker node into the GM's internal partition. It does so by assigning a logical worker node to the GM's internal partition with just the requested resources and deducting these resources from the actual node's pool of available resources. Therefore, we formally define a repartition operation on a node $N = (K, M)$ for a task $T = (R, PC)$ as the creation of a logical node $N' = (K', M)$, where $K' = R$, and updating N such that $N = (K - R, M)$. The logical node N' is added to the GM's internal partition in the LM.

This capability of Megha to place tasks anywhere in the data center using the cached global state ensures that all task placement constraints are satisfied with minimal overhead. A simplified representation of the GM's decision-making process

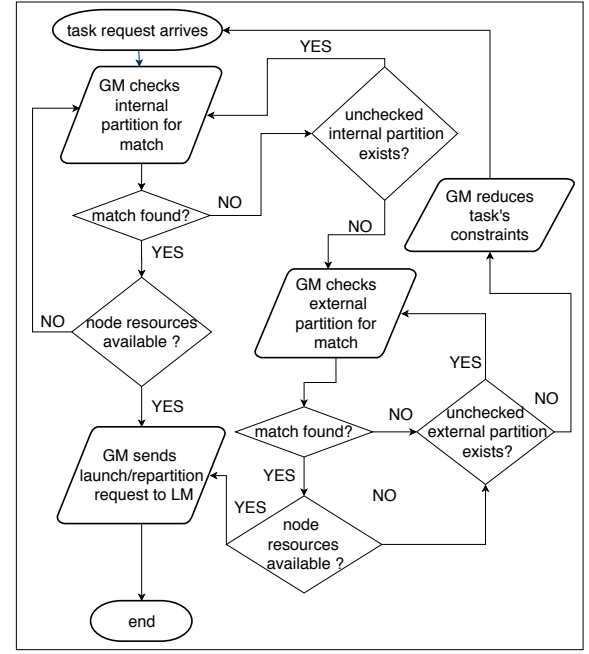


Fig. 2. Simplified representation of a GM's decision making process

is shown in Fig. 2. It is important to note that the repartitioning is performed at resource level granularity(number of vCPUs, MB of Memory) and not at the individual worker node level. This increases the probability of the framework finding a match for a task while also promoting the sharing of resources, thereby improving the utilization of the data center. To distribute the load evenly across all LMs, the GM uses round-robin while picking external partitions.

While the figure shows the movement of a worker node from one partition to another, it must be noted that this movement is implemented with minimal overhead by changing only the logical mapping of the worker node to a partition. The figure also simplifies the depiction of the repartition operation by showing it at the granularity of an entire worker node. However, Megha is capable of more fine-grained repartitioning, as mentioned earlier.

If none of the worker nodes in the partition can satisfy both the placement constraints and the resource requirements of the task, the GM moves on to the next partition and repeats the process. This continues until the GM finds an eligible worker node or all the internal partitions have been searched without finding a match. When no match is found, the GM looks at the external partitions. If an eligible worker node is found in one of the external partitions, the GM requests the LM to perform a repartition operation.

5) *Rescheduling operation:* When a task's requirements cannot be satisfied by the internal partitions of a GM or by repartitioning, the GM inserts the task request at the end of its task request queue. When the request reaches the front of the queue, the GM treats it like a new request and follows the regular decision-making process as depicted in Fig. 2.

6) *State Inconsistencies*: As mentioned earlier, the GM’s view of the state of the system, while global, maybe stale or inconsistent. This may cause the GM to make invalid launch requests to the LM requesting resources that are no longer available. There are two scenarios where the GM can make incorrect launch requests. The first is where the requested resources from the GM’s internal partitions have been assigned to another GM as a result of a repartition operation. The second scenario is where the GM makes a repartition request but the requested resources are no longer available in the external partition - either due to the resources having been consumed by the GM in charge or due to a repartition commissioned by a third GM. Both scenarios are a result of an inconsistent or stale global state in the GM. When such a request is made to the LM, the LM responds with a failure message along with the current state of all its partitions. The overhead of state inconsistencies is equal to the sum of the time taken to make a launch request to the LM, and the amount of time the LM takes to check its cluster state and respond.

C. Fairness in Megha

Megha uses queues to enforce fair sharing. Each user is assigned a queue configured with the user’s share of the data center’s resources. All tasks from the user’s jobs are submitted to this queue. Each queue is mapped to one GM only. However, a single GM can service requests from multiple queues belonging to different users. The processes the task requests from the queues in using round-robin. To ensure optimal utilization of data center resources, Megha enforces fairness only when there is resource contention. Therefore, the GM does not perform checks when a task request (with resource and placement constraints) can be satisfied by its internal partitions or by a repartition operation. Only when the GM is unable to find an eligible worker node for a task does the fairness algorithm come into play. When a match cannot be found, the GM first checks if the requesting user has already consumed her share of the resources. If not, the GM finds the user with the largest share violation and preempts the corresponding user’s tasks such that it yields the required resources. However, if the requesting user has already consumed her share, the task request is re-inserted into the user’s queue. The pseudo-code of the fairness algorithm is given in Algorithm 2.

Each GM has consistent state information about its own user queues only and has eventually-consistent (possibly stale) information about other user queues. Since preemption requests are made to preempt tasks belonging to other queues, preemption requests have to be verified by the LM. If the verification fails, the GM finds other eligible tasks for preemption and repeats the process. Since the GM has access to the consistent state of its own queues, our approach ensures against any of the queues unfairly consuming excess resources when there is resource contention. When there is resource contention, the GM checks if the user queue has already consumed its share. If it has, the task cannot request for preemption of tasks

Algorithm 2: Pseudo-code for the fairness algorithm

```

Input: task request: request
begin
  Function get_Preempt_Tasks(request)
    if share(request.user) <
      resources_consumed(user) then
      | return Failure
    else
      // Check user in decreasing order of violation
      while preempt_user = get_next_user() do
      | preempt_tasks =
      |   get_preemption_tasks(preempt_user)
      |   if preempt_tasks != empty then
      |   | return preempt_tasks
      end
      return [] // empty list
    end
  Function Main(request)
    status = Launch_Task_Internally(request)
    if status != Success then
      status = Repartition_and_Launch(request)
      if status != Success then
        tasks = get_Preempt_Tasks(request)
        if tasks empty then
        | Reinsert request in user’s task queue
        else
        | Preempt_and_Launch(tasks, request)
        end
      end
    end

```

belonging to other user queues. However, due to the GM’s inconsistent state, it may preempt tasks from queues that are no longer in violation of their share. In such a scenario, after the preempted task reaches the front of the task queue again, it is scheduled by the GM as a regular task. If sufficient resources are not available in spite of the queue adhering to its share, the GM can preempt tasks from other queues that are in violation. In addition to the heartbeats it receives, the GM updates its state regarding the resource consumption of the queues each time it interacts with an LM.

D. High Availability

The Global Masters in Megha are stateless. Each GM’s state can be reconstructed on failure from the heartbeats received from the Local Masters. Failure recovery of Local Masters and the worker nodes can be borrowed from traditional approaches. On failure, a worker can be restarted by the LM in charge; the LM must relaunch all tasks previously running on the worker with the help of its active task list. High availability of the LMs can be implemented using an active-standby configuration, such that, both the active and the standby LMs are sent all the updates from the workers and the GMs, but only the active LM enforces the GMs’ scheduling decisions.

V. EVALUATION

To demonstrate that Megha outperforms both distributed and centralized approaches, we compare its decentralized approach with Sparrow and a single-LM-single-GM configuration of Megha which behaves like a centralized scheduler. We study the performance of Megha under different scenarios with different cluster sizes and framework parameters. In all the scenarios, the heartbeat period is set to 10 seconds.

We have evaluated our framework with workloads derived from production traces. The workloads contain task resource requests, task durations, and task placement constraints. A prototype of the framework was tested on 128 cloud instances and the simulation of the framework was evaluated for data centers of sizes up to 50,000 worker nodes.

An algorithm from previous work [19] was used to model the heterogeneity of the data center and to assign various machine constraints to the worker nodes in each cluster. Each cluster has been generated with machine constraint probabilities of one of the clusters- A, B, or C, published in the work, chosen at random. Therefore, the clusters under each LM bear different distributions of the machine constraints.

A. Workloads

Our experiments were conducted using 3 workload samples- `google_19`, `alibaba` and `cloudera`. They have been derived from the production traces published by Google [22], Alibaba [1] and Cloudera[4]. The `google_19` workload consists of 4550 tasks, the `alibaba` workload consists of 8000 tasks and the `cloudera` workload consists 10173 tasks.

To evaluate the performance of Megha for tasks with placement constraints, we augmented each task in the `alibaba` and `google_19` traces with placement constraints using the methodology, and distributions published in previous work [19]. We use a sample of the `cloudera` trace to compare the performance of Sparrow and Megha under different load conditions. We use the `google_19` trace sample to demonstrate Megha's global fair scheduling.

The resource requests of the workloads have been scaled down for the experiments with the prototype. The CPU requested by the tasks have been scaled down by a factor of 400, and the memory has been scaled down by a factor of 50. However, the number of tasks, the duration of the tasks, the inter-arrival times, and the task placement constraints remain identical.

B. Simulation

For the simulation, we have considered data centers from 1000-50,000 workers. For all data center sizes, the worker nodes were given a configuration of 64 CPU cores and 16GB of RAM. The simulations were run on a 4-core Intel Xeon E5-2683 v4 machine. We use the publicly available Sparrow simulator in our work. The Sparrow simulator does not consider any overheads other than a constant network delay and queuing delays at the workers. The Megha simulator is a single-node deployment of the prototype. Therefore, unlike the Sparrow simulator, it also takes into account scheduling

overheads such as the overheads of message processing and the overheads from the decision-making incurred at each LM, GM, and worker node. The only delay that the simulator does not account for is the delay due to interference from the co-location of tasks. For a fair comparison with Megha, we run the Sparrow simulator with a modification to the network delay parameter to account for message processing overhead. We retain the default value of the network delay (0.5ms) while calculating the overhead for probing. However, we calculate the network overhead of the task information being sent to the worker differently. We replace the default network delay for the operation with the median of the transfer time for task launch messages recorded by Megha since these messages are expected to be the same in both frameworks.

C. Prototype

The prototype was deployed on 128 nodes on the Linode cloud service. Each node had 2 vCPUS and 4GB of RAM. The size of the cluster under each LM was 40 nodes. Each GM and LM ran on a separate node. The Request Generator and RabbitMQ servers were deployed on one node each.

In all scenarios, each worker node in the system has been assigned machine constraints according to the probability distributions and algorithms mentioned earlier.

D. Implementation

Megha uses HTTP REST APIs for synchronous communication such as launch, repartition, and preemption requests, and RabbitMQ messages for asynchronous updates. Both the simulation and the prototype have been implemented using Python3.8. Gunicorn 20.0.4 has been used as the HTTP server in all the components. RabbitMQ v3.8.5 server was used for communication.

VI. RESULTS

A. Comparison with distributed and Centralized schedulers

We compare the resource allocation times recorded for Sparrow and Megha for 10k workers in Fig. 3 and 4. The resource consumption curve of the Cloudera workload is plotted against time in Fig. 5. The median allocation time of Sparrow is smaller than Megha's. However, it can be observed in the graphs that Megha's allocation time is consistently less than 1.5s, whereas the allocation times recorded under Sparrow have a large variation and can run into tens of seconds (max: 100s). The long tail in Sparrow's distribution can be attributed to its inability to ascertain which nodes are available using random-sampling.

We also compare the resource allocation times reported by Megha's decentralized configuration and its centralized configuration (single-LM-single-GM). We use the placement constraints-augmented `alibaba` trace for this purpose. The results are shown in Fig. 6. Due to the decrease in parallelism in the GMs and LMs, the centralized case reports a very high allocation time when compared to Megha.

B. Component Delays

We plotted Megha's component delays (Fig. 7) described in Section III taking the median of the delays recorded for the simulation runs with the Cloudera trace. The results show that the largest contributor is the *Communication_Delay* component which contributes to over 90% of the delay. The *Processing_Delay* components at the LMs and GMs contribute to less than 3% of the allocation time. The *Framework_Queueing_Delay* constitutes $\sim 7\%$ of the allocation time.

C. Cluster Size

We study the impact of data center size on Megha's allocation time keeping the number of LMs and GMs constant at 10. Fig. 8, shows how Megha's resource allocation time increases with an increase in the number of workers when the number of LMs and GMs are not correspondingly increased. This delay can be attributed to the increase in the amount of global state, and the size of each LM update that a GM must process as the number of workers increases.

D. Number of LMs and GMs

We study the impact of the number of LMs and GMs on the task allocation time, using simulations run with the *alibaba* trace for a data center of size 30,000 worker nodes (Fig. 9). Increasing the number of GMs increases the scheduling throughput of the system, thereby decreasing the *Framework_Queueing_Delay* experienced by each task. Increasing the number of LMs increases the total number of GM requests that can be processed per second. It also reduces the size of the updates sent to the GMs, resulting in smaller processing overheads per request. However, when the number of GMs exceeds the number of LMs, we find that there is a slight increase in the allocation times reported. This can be attributed to the LMs being unable to keep up with the throughput of the GMs resulting in a larger *Framework_Queueing_Delay* at the LMs.

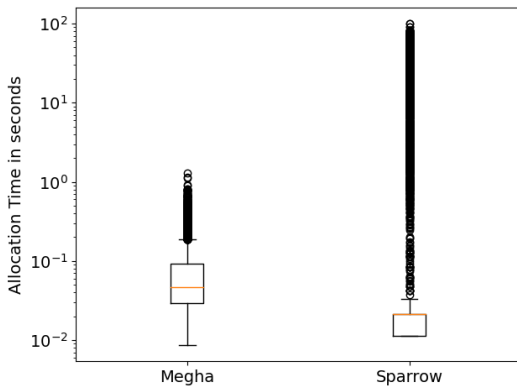


Fig. 3. Distribution of allocation times reported by Megha and Sparrow

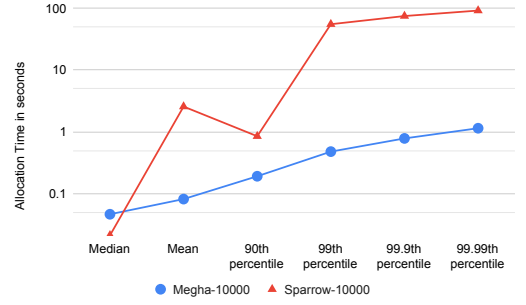


Fig. 4. Median, Mean, 90th percentile, 99th percentile, 99.9th percentile and 99.99th percentile Allocation time recorded for Megha and Sparrow

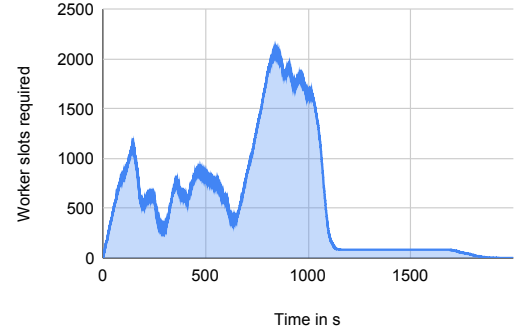


Fig. 5. Total number of worker slots required by the workload sample

E. Fairness

Megha's fairness algorithm only kicks in when there is resource contention amongst the users. Therefore, we chose smaller data center sizes to encourage resource contention. Fig. 10 shows the allocation times of tasks when the simulation is run for data centers of sizes 200, 500, and 1500. We configured Megha with 4 user queues having 10%, 25%, 15%, and 50% resource shares. The first user queue is mapped to GM 1, the second and third to GM 2, and the last to GM 3. We have used the *google_19* trace for this analysis. The number

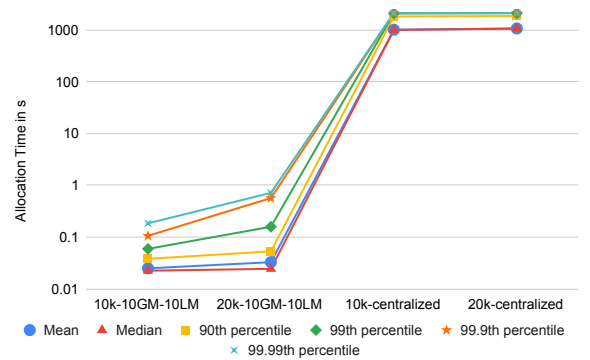


Fig. 6. Comparison of allocation time for 10k and 20k worker nodes for centralized and decentralized approaches

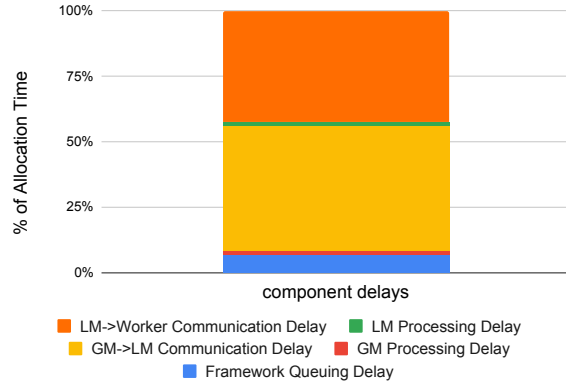


Fig. 7. Contributions of the Component Delays

of repartitions, preemption attempts, and task preemptions is shown in Fig. 11. The allocation time is seen to increase with a decrease in the cluster size. This is because, with a decrease in the cluster size, resources are scarcer and tasks are made to wait longer for the required resources to free up. The number of preemptions, preemption attempts, and repartitions also follow the same trend and is found to increase with the decrease in cluster size.

F. Performance Evaluation with Prototype

The distribution of the allocation time of the prototype evaluated on Linode instances is shown in Fig. 12. The allocation times recorded in the prototype were found to be higher than those recorded during the simulation. This can be explained by the delays observed in the tasks' execution on each worker node. The median execution delays recorded were 3.32 s for *google_19* and 2.15s for *alibaba*. There is no execution delay recorded in the simulation results because the simulator does not account for interference. The prototype, on the other hand, runs the task such that the task completes its execution only after it finishes consuming the requested resources for the task's duration. That is, the task requests 0.5

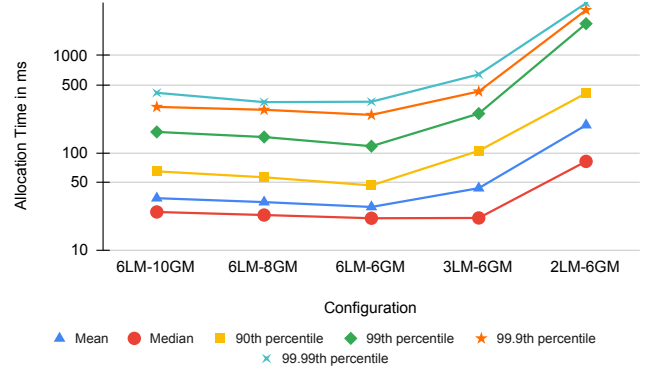


Fig. 9. Allocation time recorded for different configurations

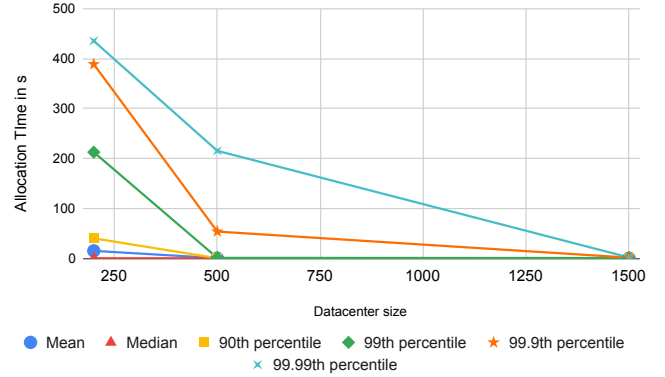


Fig. 10. Allocation time recorded for different data center sizes with fairness

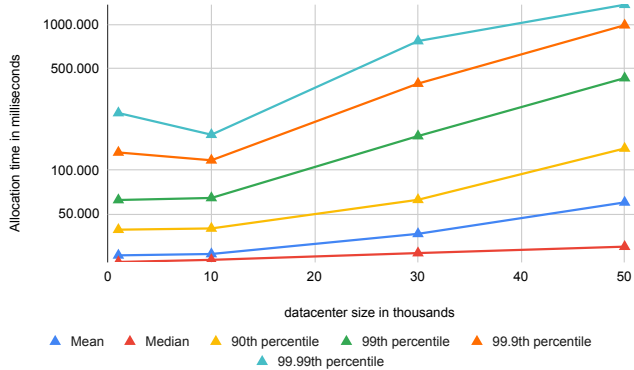


Fig. 8. Allocation time recorded for different data center sizes with 10LM-10GM configuration

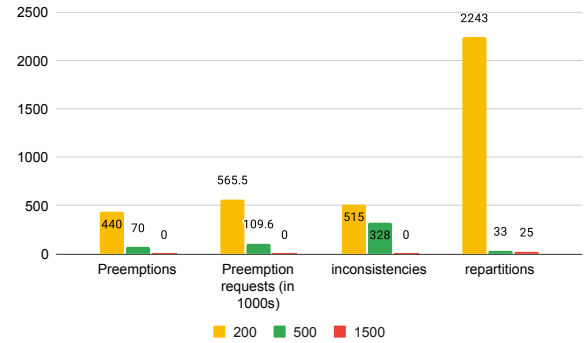


Fig. 11. Number of preemptions, repartitions and preemption attempts recorded for each data center size

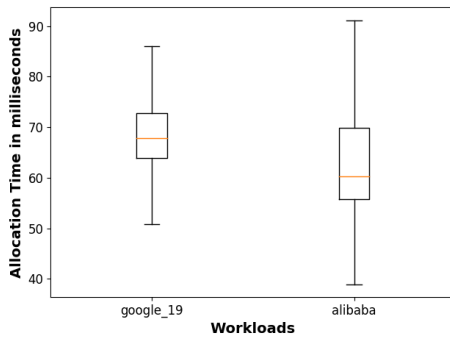


Fig. 12. Distribution of the allocation time recorded for the workloads scheduled using the prototype

CPU cores for 2 seconds, the task is said to have completed its execution only after consuming 1 CPU-core-second on the worker node. Due to interference on worker nodes with co-located tasks, the tasks need to run for longer than the duration specified to consume the required resources. This results in the resources being held for longer, which consequently leads to larger allocation times in tasks that require resources on the same worker node. The load from the google_19 workload is higher, causing greater resource contention and interference thereby leading to larger delays in the task completion times.

VII. CONCLUSION

Data center resource management and scheduling frameworks have received a lot of attention in the recent past. The frameworks must be able to ensure optimal performance of workloads and at the same time, guarantee low-latency allocation decisions. This paper describes a decentralized resource management framework for federated clusters, named Megha. The framework uses a flexible partitioning scheme and eventual-consistency to achieve scalability and low allocation times while satisfying the various placement constraints of the tasks and guaranteeing global fairness. Our experiments conducted with workloads derived from 3 production traces, and with different data center sizes ranging from 120 to 50k nodes, show that Megha can achieve median allocation times comparable to that of distributed schedulers (Sparrow), and improves on their 99th percentile tail latency by 2 orders of magnitude.

VIII. FUTURE WORK

We are currently working on two extensions to the study presented in the paper. The first is to evaluate the impact of varying the heartbeat period on the overall performance of the system. We also wish to extend Megha to support scheduling policies such as gang scheduling, and priority-based scheduling.

REFERENCES

- [1] *Alibaba Cluster Trace v2018*. URL: <https://github.com/alibaba/clusterdata/tree/master/cluster-trace-v2018>.
- [2] George Amvrosiadis et al. “On the diversity of cluster workloads and its impact on research results”. In: *2018 {USENIX} Annual Technical Conference*. 2018, pp. 533–546.
- [3] Eric Boutin et al. “Apollo: Scalable and coordinated scheduling for cloud-scale computing”. In: *11th {USENIX} Symposium on Operating Systems Design and Implementation ({OSDI} 14)*. 2014, pp. 285–300.
- [4] Yanpei Chen, Sara Alspaugh, and Randy Katz. “Interactive analytical processing in big data systems: A cross-industry study of mapreduce workloads”. In: *arXiv preprint arXiv:1208.4174* (2012).
- [5] Yue Cheng, Zheng Chai, and Ali Anwar. “Characterizing co-located datacenter workloads: An alibaba case study”. In: *Proceedings of the 9th Asia-Pacific Workshop on Systems*. 2018, pp. 1–3.
- [6] Carlo Curino et al. “Hydra: a federated resource manager for data-center scale analytics”. In: *16th {USENIX} Symposium on Networked Systems Design and Implementation ({NSDI} 19)*. 2019, pp. 177–192.
- [7] Pamela Delgado et al. “Hawk: Hybrid datacenter scheduling”. In: *2015 {USENIX} Annual Technical Conference*. 2015, pp. 499–510.
- [8] Pamela Delgado et al. “Job-aware scheduling in eagle: Divide and stick to your probes”. In: *Proceedings of the Seventh ACM Symposium on Cloud Computing*. 2016, pp. 497–509.
- [9] Christina Delimitrou and Christos Kozyrakis. “Paragon: QoS-aware scheduling for heterogeneous datacenters”. In: *ACM SIGPLAN Notices* 48.4 (2013), pp. 77–88.
- [10] Christina Delimitrou and Christos Kozyrakis. “Quasar: resource-efficient and QoS-aware cluster management”. In: *ACM SIGPLAN Notices* 49.4 (2014), pp. 127–144.
- [11] Jing Guo et al. “Who limits the resource efficiency of my datacenter: An analysis of alibaba datacenter traces”. In: *2019 IEEE/ACM 27th International Symposium on Quality of Service (IWQoS)*. IEEE. 2019, pp. 1–10.
- [12] Chunliang Hao et al. “PCSampler: Sample-based, Private-state Cluster Scheduling”. In: *2017 17th IEEE/ACM International Symposium on Cluster, Cloud and Grid Computing (CCGRID)*. IEEE. 2017, pp. 599–608.
- [13] Mohammad Jassas and Qusay H Mahmoud. “Failure analysis and characterization of scheduling jobs in google cluster trace”. In: *IECON 2018-44th Annual Conference of the IEEE Industrial Electronics Society*. IEEE. 2018, pp. 3102–3107.
- [14] Konstantinos Karanasos et al. “Mercury: Hybrid centralized and distributed scheduling in large shared clusters”. In: *2015 {USENIX} Annual Technical Conference*. 2015, pp. 485–497.
- [15] Chengzhi Lu et al. “Imbalance in the cloud: An analysis on alibaba cluster trace”. In: *2017 IEEE International Conference on Big Data (Big Data)*. IEEE. 2017, pp. 2884–2892.

- [16] Kay Ousterhout et al. “Sparrow: distributed, low latency scheduling”. In: *Proceedings of the Twenty-Fourth ACM Symposium on Operating Systems Principles*. 2013, pp. 69–84.
- [17] Charles Reiss et al. “Heterogeneity and dynamicity of clouds at scale: Google trace analysis”. In: *Proceedings of the Third ACM Symposium on Cloud Computing*. 2012, pp. 1–13.
- [18] Francisco Romero and Christina Delimitrou. “Mage: Online and interference-aware scheduling for multi-scale heterogeneous systems”. In: *Proceedings of the 27th International Conference on Parallel Architectures and Compilation Techniques*. 2018, pp. 1–13.
- [19] Bikash Sharma et al. “Modeling and synthesizing task placement constraints in Google compute clusters”. In: *Proceedings of the 2nd ACM Symposium on Cloud Computing*. 2011, pp. 1–14.
- [20] Vinod Kumar Vavilapalli et al. “Apache hadoop yarn: Yet another resource negotiator”. In: *Proceedings of the 4th annual Symposium on Cloud Computing*. 2013, pp. 1–16.
- [21] Zhijun Wang et al. “Pigeon: an Effective Distributed, Hierarchical Datacenter Job Scheduler”. In: *Proceedings of the ACM Symposium on Cloud Computing*. 2019, pp. 246–258.
- [22] J Wilkes. *Google cluster-usage traces v3*. Tech. rep. Technical report at <https://github.com/google/cluster-data>, Google ..., 2019.
- [23] Hailong Yang et al. “Bubble-flux: Precise online qos management for increased utilization in warehouse scale computers”. In: *ACM SIGARCH Computer Architecture News* 41.3 (2013), pp. 607–618.